Parallel OWL Reasoning: Merge Classification

Kejia Wu(✉) and Volker Haarslev

Department of Computer Science and Software Engineering,
Concordia University, Montreal, Canada
w_kejia@cs.concordia.ca

Abstract. Our research is motivated by the ubiquitous availability of multiprocessor computers and the observation that available *Web Ontology Language (OWL)* reasoners only make use of a single processor. This becomes rather frustrating for users working in ontology development, especially if their ontologies are complex and require long processing times using these OWL reasoners. We present a novel algorithm that uses a *divide and conquer* strategy for parallelizing OWL TBox classification, a key task in description logic reasoning. We discuss some interesting properties of our algorithm, e.g., its suitability for distributed reasoning, and present an empirical study using a set of benchmark ontologies, where a speedup of up to a factor of 4 has been observed when using 8 workers in parallel.

1 Introduction

Due to the semantic web, a multitude of ontologies are emerging. Quite a few ontologies are huge and contain hundreds of thousands of concepts. Although some of these huge ontologies fit into one of OWL's three tractable profiles, e.g., the well known Snomed ontology is in the $\mathcal{EL}$ profile, there still exist a variety of other OWL ontologies that make full use of OWL DL and require long processing times, even when highly optimized OWL reasoners are employed. Moreover, although most of the huge ontologies are currently restricted to one of the tractable profiles in order to ensure fast processing, it is foreseeable that some of them will require an expressivity that is outside of the tractable OWL profiles.

The research presented in this paper is targeted to provide better OWL reasoning scalability by making efficient use of modern hardware architectures such as multi-processor/core computers. This becomes more important in the case of ontologies that require long processing times although highly optimized OWL reasoners are already used. We consider our research an important basis for the design of next-generation OWL reasoners that can efficiently work in a parallel/concurrent or distributed context using modern hardware. One of the major obstacles that need to be addressed in the design of corresponding algorithms and architectures is the overhead introduced by concurrent computing and its impact on scalability.

Traditional divide and conquer algorithms split problems into independent sub-problems before solving them under the premise that not much

DOI: 10.1007/978-3-319-06826-8_17, © Springer International Publishing Switzerland 2014

communication among the divisions is needed when independently solving the sub-problems, so shared data is excluded to a great extent. Therefore, divide and conquer algorithms are in principle suitable for concurrent computing, including shared-memory parallelization and distributed systems.

Furthermore, recently research on *ontology partitioning* has been proposed and investigated for dealing with monolithic ontologies. Some research results, e.g. ontology modularization [10], can be used for decreasing the scale of an ontology-reasoning problem. The reasoning over a set of sub-ontologies can be executed in parallel. However, there is still a solution needed to reassemble sub-ontologies together. The algorithms presented in this paper can also serve as a solution for this problem.

In the remaining sections, we present our merge-classification algorithm which uses a divide and conquer strategy and a heuristic partitioning scheme. We report on our conducted experiments and their evaluation, and discuss related research.

2 A Parallelized Merge Classification Algorithm

In this section, we present an algorithm for classifying *Description Logic (DL)* ontologies. Due to lack of space we refer for preliminaries about DLs, DL reasoning, and semantic web to [3,13].

We present the merge-classification algorithm in pseudo code. Part of the algorithm is based on standard top- and bottom-search techniques to incrementally construct the classification hierarchy (e.g., see [2]). Due to the symmetry between *top-down* ($\top$*_search*) and *bottom-up* ($\bot$*_search*) search, we only present the first one. In the pseudo code, we use the following notational conventions: Δ_i, Δ_α, and Δ_β designate sub-domains that are divided from Δ; we consider a subsumption hierarchy as a partially order over Δ, denoted as $\leq$, a subsumption relationship where C is subsumed by D ($C \sqsubseteq D$) is expressed by $C \leq D$ or by $\langle C, D \rangle \in \leq$, and $\leq_i$, $\leq_\alpha$, and $\leq_\beta$ are subsumption hierarchies over Δ_i, Δ_α, and Δ_β, respectively; in a subsumption hierarchy over Δ, $C \prec D$ designates $C \sqsubseteq D$ and there does not exist a named concept E such that $C \leq E$ and $E \leq D$; $\prec_i$, $\prec_\alpha$ and $\prec_\beta$ are similar notations defined over Δ_i, Δ_α, and Δ_β, respectively.

Our merge-classification algorithm classifies a taxonomy by calculating its divided sub-domains and then by merging the classified sub-taxonomies together. The algorithm makes use of two facts: (i) If it holds that $B \leq A$, then the subsumption relationships between B's descendants and A's ancestors are determined; (ii) if it is known that $B \not\leq A$, the subsumption relationships between B's descendants and A's ancestors are undetermined. The canonical DL classification algorithms, top-search and bottom-search, have been modified and integrated into the merge-classification. The algorithm consists of two stages: divide and conquering, and combining. Algorithm 1 shows the main part of our parallelized DL classification procedure. The keyword *spawn* indicates that its following calculation must be executed in parallel, either creating a new thread in a shared-memory context or generating a new process or session in a non-shared-memory context. The keyword *sync* always follows *spawn* and suspends the current calculation procedure until all calculations invoked by *spawn* return.

Algorithm 1. $\kappa(\Delta_i)$

```
input  : The sub-domain $\Delta_i$
output: The subsumption hierarchy classified over $\Delta_i$
1  begin
2      if $divided\_enough?(\Delta_i)$ then
3          return $classify(\Delta_i)$;
4      else
5          $\langle \Delta_\alpha, \Delta_\beta \rangle \leftarrow divide(\Delta_i)$;
6          $\leq_\alpha \leftarrow$ spawn $\kappa(\Delta_\alpha)$;
7          $\leq_\beta \leftarrow \kappa(\Delta_\beta)$;
8          sync;
9          return $\mu(\leq_\alpha, \leq_\beta)$;
10     end if
11 end
```

Algorithm 2. $\mu(<_\alpha, <_\beta)$

```
input  :  The master subsumption hierarchy $\leq_\alpha$
          The subsumption hierarchy $\leq_\beta$ to be merged into $\leq_\alpha$
output:  The subsumption hierarchy resulting from merging $\leq_\alpha$ over $\leq_\beta$
1  begin
2      $\top_\alpha \leftarrow select\text{-}top(\leq_\alpha)$;
3      $\top_\beta \leftarrow select\text{-}top(\leq_\beta)$;
4      $\bot_\alpha \leftarrow select\text{-}bottom(\leq_\alpha)$;
5      $\bot_\beta \leftarrow select\text{-}bottom(\leq_\beta)$;
6      $\leq_\alpha \leftarrow \top\_merge(\top_\alpha, \top_\beta)$;
7      $\leq_i \leftarrow \bot\_merge(\bot_\alpha, \bot_\beta)$;
8      return $\leq_i$;
9  end
```

The domain Δ is divided into smaller partitions in the first stage. Then, classification computations are executed over each sub-domain Δ_i. A classified sub-terminology $\leq_i$ is inferred over Δ_i. This divide and conquering operations can progress in parallel.

Classified sub-terminologies are to be merged in the combining stage. The told subsumption relationships are utilized in the merging process. Algorithm 2 outlines the master procedure, and the slave procedure is addressed by Algorithms 3, 4, 5, and 6.

2.1 Divide and Conquer Phase

The first task is to divide the universe, Δ, into sub-domains. Without loss of generality, Δ only focuses on *significant* concepts, i.e., concept names or atomic concepts, that are normally declared explicitly in some ontology $\mathcal{O}$, and *intermediate* concepts, i.e., non-significant ones, only play a role in subsumption

Algorithm 3. $\top\_search(C, D, \leq_i)$

```
input  : $C$: the new concept to be classified
         $D$: the current concept with $\langle D, \top \rangle \in \leq_i$
         $\leq_i$: the subsumption hierarchy
output: The set of parents of $C$: $\{p \mid \langle C, p \rangle \in \leq_i\}$.
begin
    $mark\_visited(D)$;
    $green \leftarrow \phi$;
    forall the $d \in \{d \mid \langle d, D \rangle \in \prec_i\}$ do /* collect all children of $D$ that subsume $C$ */
        if $\leq ?(C, d)$ then
            $green \leftarrow green \cup \{d\}$;
        end if
    end forall
    $box \leftarrow \phi$;
    if $green = \phi$ then
        $box \leftarrow \{D\}$;
    else
        forall the $g \in green$ do
            if $\neg marked\_visited?(g)$ then
                $box \leftarrow box \cup \top\_search(C, g, \leq_i)$ ; /* recursively test whether $C$ is
                subsumed by the descendants of $g$ */
            end if
        end forall
    end if
    return $box$; /* return the parents of $C$ */
end
```

tests. Each sub-domain is classified independently. The *divide* operation can be naively implemented as an even partitioning over Δ, or by more sophisticated clustering techniques such as *heuristic partitioning* that may result in a better performance, as presented in Sect. 3. The conquering operation can be any standard DL classification method. We first present the most popular classification methods, top-search (Algorithm 3) and bottom-search (omitted here).

The DL classification procedure determines the most specific super- and the most general sub-concepts of each significant concept in Δ. The classified concept hierarchy is a partial order, $\leq$, over Δ. $\top\_search$ recursively calculates a concept's *intermediate* predecessors, i.e., intermediate immediate ancestors, as a relation $\prec_i$ over $\leq_i$.

2.2 Combining Phase

The independently classified sub-terminologies must be merged together in the combining phase. The original top-search (Algorithm 3) (and bottom-search) have been modified to merge two sub-terminologies $\leq_\alpha$ and $\leq_\beta$. The basic idea is to iterate over Δ_β and to use top-search (and bottom-search) to insert each element of Δ_β into $\leq_\alpha$, as shown in Algorithm 4.

Algorithm 4. $\top\_merge^{-}(A, B, \leq_\alpha, \leq_\beta)$

```
input  : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top \rangle \in \leq_\alpha$
         B: the new concept from the merged subsumption hierarchy, i.e. $\langle B, \top \rangle \in \leq_\beta$
         $\leq_\alpha$: the master subsumption hierarchy
         $\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output: The merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$.
1  begin
2      $parents \leftarrow \top\_search(B, A, \leq_\alpha)$;
3      forall the $a \in parents$ do
4          $\leq_\alpha \leftarrow \leq_\alpha \cup \langle B, a \rangle$; /* insert B into $\leq_\alpha$ */
5          forall the $b \in \{b \mid \langle b, B \rangle \in \prec_\beta\}$ do /* insert children of B (in $\leq_\beta$) below parents
           of B (in $\leq_\alpha$) */
6              $\leq_\alpha \leftarrow \top\_merge^{-}(a, b, \leq_\alpha, \leq_\beta)$;
7          end forall
8      end forall
9      return $\leq_\alpha$;
10 end
```

Algorithm 5. $\top\_merge(A, B, \leq_\alpha, \leq_\beta)$

```
input  : A: the current concept of the master subsumption hierarchy, i.e. $\langle A, \top \rangle \in \leq_\alpha$
         B: the new concept of the merged subsumption hierarchy, i.e. $\langle B, \top \rangle \in \leq_\beta$
         $\leq_\alpha$: the master subsumption hierarchy
         $\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
output: the merged subsumption hierarchy $\leq_\alpha$ over $\leq_\beta$
1  begin
2      $parents \leftarrow \top\_search^*(B, A, \leq_\beta, \leq_\alpha)$;
3      forall the $a \in parents$ do
4          $\leq_\alpha \leftarrow \leq_\alpha \cup \langle B, a \rangle$;
5          forall the $b \in \{b \mid \langle b, B \rangle \in \prec_\beta\}$ do
6              $\leq_\alpha \leftarrow \top\_merge(a, b, \leq_\alpha, \leq_\beta)$;
7          end forall
8      end forall
9      return $\leq_\alpha$;
10 end
```

However, this method does not make use of so-called told subsumption (and non-subsumption) information contained in the merged sub-terminology $\leq_\beta$. For example, it is unnecessary to test $\leq ?(B_2, A_1)$ when we know $B_1 \leq A_1$ and $B_2 \leq B_1$, given that A_1, A_2 occur in Δ_α and B_1, B_2 occur in Δ_β.

Therefore, we designed a novel algorithm in order to utilize the properties addressed by Proposition 1 to 6. The calculation starts top-merge (Algorithm 5), which uses a modified top-search algorithm (Algorithm 6). This pair of procedures find the most specific subsumers in the master sub-terminology $\leq_\alpha$ for every concept from the sub-terminology $\leq_\beta$ that is being merged into $\leq_\alpha$.

Proposition 1. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A \rangle \in \prec_i$ is found in top-search, $\langle A, \top \rangle \in \leq_\alpha$ and $\langle B, \top \rangle \in \leq_\beta$, then for $\forall b \in \{b \mid \langle b, B \rangle \in \leq_\beta\}$ and $\forall a \in \{a \mid \langle A, a \rangle \in \leq_\alpha\} \cup \{A\}$ it is unnecessary to calculate whether $b \leq a$.*

Algorithm 6. $\top\_search^*(C, D, \leq_\beta, \leq_\alpha)$

```
input  : $C$: the new concept to be inserted into $\leq_\alpha$, and $\langle C, \top \rangle \in \leq_\beta$
         $D$: the current concept, and $\langle D, \top \rangle \in \leq_\alpha$
         $\leq_\beta$: the subsumption hierarchy to be merged into $\leq_\alpha$
         $\leq_\alpha$: the master subsumption hierarchy
output: The set of parents of $C$: $\{p \mid \langle C, p \rangle \in \leq_\alpha\}$
begin
    $mark\_visited(D)$;
    $green \leftarrow \phi$; /* subsumers of $C$ that are from $\leq_\alpha$ */
    $red \leftarrow \phi$; /* non-subsumers of $C$ that are children of $D$ */
    forall the $d \in \{d \mid \langle d, D \rangle \in \prec_\alpha \wedge \langle d, \top \rangle \notin \leq_\beta\}$ do
        if $\leq ?(C, d)$ then
            $green \leftarrow green \cup \{d\}$;
        else
            $red \leftarrow red \cup \{d\}$;
        end if
    end forall
    $box \leftarrow \phi$;
    if $green = \phi$ then
        if $\leq ?(C, D)$ then
            $box \leftarrow \{D\}$;
        else
            $r \leftarrow \{D\}$;
        end if
    else
        forall the $g \in green$ do
            if $\neg marked\_visited?(g)$ then
                $box \leftarrow box \cup \top\_search^*(C, g, \leq_\beta, \leq_\alpha)$;
            end if
        end forall
    end if
    forall the $r \in red$ do
        forall the $c \in \{c \mid \langle c, C \rangle \in \prec_i\}$ do
            $\leq_\alpha \leftarrow \top\_merge(r, c, \leq_\alpha, \leq_\beta)$;
        end forall
    end forall
    return $box$;
end
```

Proposition 2. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle B, A \rangle \in \prec_i$ is found in top-search, $\langle A, \top \rangle \in \leq_\alpha$ and $\langle B, \top \rangle \in \leq_\beta$, then for $\forall b \in \{b \mid \langle b, B \rangle \in \prec_\beta \wedge b \neq B\}$ and $\forall a \in \{a \mid \langle a, A \rangle \in \prec_\alpha \wedge a \neq A\}$ it is necessary to calculate whether $b \leq a$.*

Proposition 3. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $B \not\leq A$ is found in top-search, $\langle A, \top \rangle \in \leq_\alpha$ and $\langle B, \top \rangle \in \leq_\beta$, then for $\forall b \in \{b \mid \langle b, B \rangle \in \leq_\beta \wedge b \neq B\}$ and $\forall a \in \{a \mid \langle a, A \rangle \in \leq_\alpha\} \cup \{A\}$ it is necessary to calculate whether $b \leq a$.*

Proposition 4. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle A, B \rangle \in \prec_i$ is found in bottom-search, $\langle \bot, A \rangle \in \leq_\alpha$ and $\langle \bot, B \rangle \in \leq_\beta$, then for $\forall b \in \{b \mid$*

$\langle B, b\rangle \in \leq_\beta\}$ and $\forall a \in \{a \mid \langle a, A\rangle \in \leq_\alpha\} \cup \{A\}$ it is unnecessary to calculate whether $a \leq b$.

Proposition 5. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $\langle A, B\rangle \in \prec_i$ is found in bottom-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b \in \{b \mid \langle B, b\rangle \in \prec_\beta \wedge b \neq B\}$ and $\forall a \in \{a \mid \langle A, a\rangle \in \prec_\alpha \wedge a \neq A\}$ it is necessary to calculate whether $a \leq b$.*

Proposition 6. *When merging sub-terminology $\leq_\beta$ into $\leq_\alpha$, if $A \not\leq B$ is found in bottom-search, $\langle \bot, A\rangle \in \leq_\alpha$ and $\langle \bot, B\rangle \in \leq_\beta$, then for $\forall b \in \{b \mid \langle B, b\rangle \in \leq_\beta \wedge b \neq B\}$ and $\forall a \in \{a \mid \langle A, a\rangle \in \leq_\alpha\} \cup \{A\}$ it is necessary to calculate whether $a \leq b$.*

When merging a concept B, $\langle B, \top\rangle \in \leq_\beta$, the top-merge algorithm first finds for B the most specific position in the master sub-terminology $\leq_\alpha$ by means of *top-down* search. When such a most specific super-concept is found, this concept and all its super-concepts are naturally super-concepts of every sub-concept of B in the sub-terminology $\leq_\beta$, as is stated by Proposition 1. However, this newly found predecessor of B may not be necessarily a predecessor of some descendant of B in $\leq_\beta$. Therefore, the algorithm continues to find the most specific positions for all sub-concepts of B in $\leq_\beta$ according to Proposition 2. Algorithm 5 addresses this procedure.

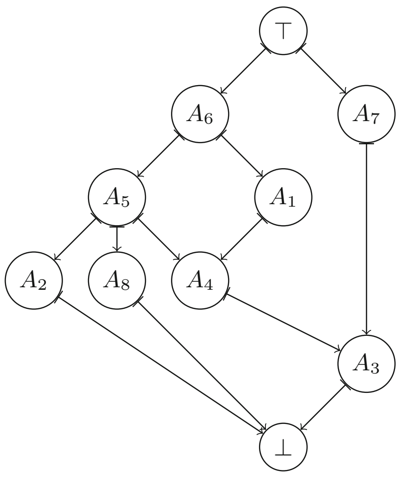
Non-subsumption information can be told in the top-merge phase. Top-down search employed by top-merge must do subsumption tests somehow. In a canonical top-search procedure, as indicated by Algorithm 3, the branch search is stopped at this point. However, the conclusion that a merged concept B, $\langle B, \top\rangle \in \leq_\beta$, is not subsumed by a concept A, $\langle A, \top\rangle \in \leq_\alpha$, does not rule out the possibility of $b \leq A$, $b \in \{b \mid \langle b, B\rangle \in \prec_\beta\}$, which is not required in traditional top-search and may be abound in the top-merge procedure, and therefore must be followed by determining whether $b \leq A$. Otherwise, the algorithm is incomplete. Proposition 3 presents this observation. For this reason, the original top-search algorithm must be adapted to the new situation. Algorithm 6 is the updated version of the top-search procedure.

Algorithm 6 not only maintains told subsumption information by the set *green*, but also propagates told non-subsumption information by the set *red* for further inference. As addressed by Proposition 3, when the position of a merged concept is determined, the subsumption relations between its successors and the *red* set are calculated. Furthermore, the subsumption relation for the concept C and D in Algorithm 6 must be explicitly calculated even when the set *green* is empty. In the original top-search procedure (Algorithm 3), $C \prec_i D$ is implicitly derivable if the set *green* is empty, which does not hold in the modified top-search procedure (Algorithm 6) since it does not always start from $\top$ any more when searching for the most specific position of a concept.

$\mathcal{T}$:

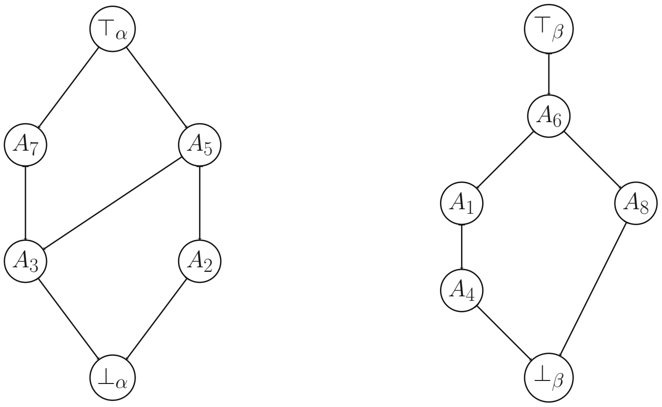
$A_6 \sqsubseteq \top$	$A_4 \sqsubseteq A_1$
$A_7 \sqsubseteq \top$	$A_3 \sqsubseteq A_7$
$A_5 \sqsubseteq A_6$	$A_3 \sqsubseteq A_4$
$A_1 \sqsubseteq A_6$	$\bot \sqsubseteq A_2$
$A_2 \sqsubseteq A_5$	$\bot \sqsubseteq A_8$
$A_8 \sqsubseteq A_5$	$\bot \sqsubseteq A_3$
$A_4 \sqsubseteq A_5$	

(a) The TBox given.



(b) The classified terminology hierarchy.

Fig. 1. An example ontology.



(a) The subsumption hierarchy $\leq_\alpha$. (b) The subsumption hierarchy $\leq_\beta$.

Fig. 2. The subsumption hierarchy over divisions.

2.3 Example

We use an example to illustrate the algorithm further. Given an ontology with a TBox defined by Fig. 1(a), which only contains simple concept subsumption axioms, Fig. 1(b) shows the subsumption hierarchy.

Suppose that the ontology is clustered into two groups in the divide phase: $\Delta_\alpha = \{A_2, A_3, A_5, A_7\}$ and $\Delta_\beta = \{A_1, A_4, A_6, A_8\}$. They can be classified independently, and the corresponding subsumption hierarchies are shown in Fig. 2.

$$\begin{tikzcd}
\top\_merge(\top_\alpha, \top_\beta, \leq_\alpha, \leq_\beta) \arrow[d] \\
\top\_search^*(\top_\beta, \top_\alpha, \leq_\beta, \leq_\alpha) \arrow[d] \arrow[u, bend right, "\{\top_\alpha\}"'] \\
\top\_merge(A_5, A_6, \leq_\alpha, \leq_\beta) \arrow[d] \arrow[u, bend right, "\leq_\alpha \leftarrow \leq_\alpha \cup \phi"'] \\
\top\_search^*(A_6, A_5, \leq_\beta, \leq_\alpha) \arrow[d] \arrow[u, bend right, "\phi"'] \\
\top\_merge(A_5, A_1, \leq_\alpha, \leq_\beta) \arrow[d] \arrow[u, bend right, "\leq_\alpha \leftarrow \leq_\alpha \cup \phi"'] \\
\top\_search^*(A_1, A_5, \leq_\beta, \leq_\alpha) \arrow[d] \arrow[u, bend right, "\phi"'] \\
\top\_merge(A_5, A_4, \leq_\alpha, \leq_\beta) \arrow[d] \arrow[u, bend right, "\leq_\alpha \leftarrow \leq_\alpha \cup \{A_4 \leq A_5\}"'] \\
\top\_search^*(A_4, A_5, \leq_\beta, \leq_\alpha) \arrow[d] \arrow[u, bend right, "\{A_5\}"'] \\
\top\_merge(A_2, \bot_\beta, \leq_\alpha, \leq_\beta) \arrow[d] \arrow[u, bend right, "\leq_\alpha \leftarrow \leq_\alpha \cup \{\bot_\beta \leq A_2\}"'] \\
\top\_search^*(\bot_\beta, A_2, \leq_\beta, \leq_\alpha) \arrow[u, bend right, "\{A_2\}"'] \\
\vdots
\end{tikzcd}$$

Fig. 3. The computation path of determining $A_4 \leq_i A_5$.

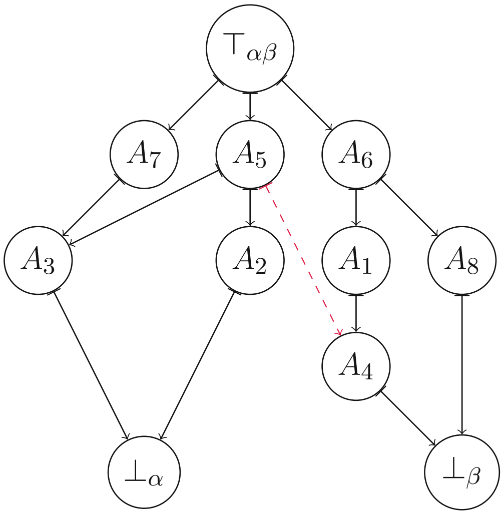


Fig. 4. The subsumption hierarchy after $A_4 \leq A_5$ has been determined.

Algorithm 7. *cluster*(G)

```
input : $G$: the subsumption graph
output: $R$: the concept names partitions
begin
    $R \leftarrow \phi$;
    $visited \leftarrow \phi$;
    $N \leftarrow get\_children(\top, G)$;
    foreach $n \in N$ do
        $P \leftarrow \{n\}$;
        $visited \leftarrow visited \cup \{n\}$;
        $R \leftarrow R \cup \{build\_partition(n, visited, G, P)\}$;
    end foreach
    return $R$;
end
```

In the merge phase, the concepts from $\leq_\beta$ are merged into $\leq_\alpha$. For example, Fig. 3 shows a possible computation path where $A_4 \leq A_5$ is being determined.[1] If we assume a subsumption relationship between two concepts is proven when the parent is added to the set *box* (see Line 15, Algorithm 6), Fig. 4 shows the subsumption hierarchy after $A_4 \leq A_5$ has been determined.

3 Partitioning

Partitioning is an important part of this algorithm. It is the main task in the dividing phase. In contrast to simple problem domains such as sorting integers, where the merge phase of a standard merge-sort does not require another sorting, DL ontologies might entail numerous subsumption relationships among concepts. Building a terminology with respect to the entailed subsumption hierarchy is the primary function of DL classification. We therefore assumed that some heuristic partitioning schemes that make use of known subsumption relationships may improve reasoning efficiency by requiring a smaller number of subsumption tests, and this assumption has been proved by our experiments, which are described in Sect. 4.

So far, we have presented an ontology partitioning algorithm by using only told subsumption relationships that are directly derived from concept definitions and axiom declarations. Any concept that has at least one told super- and one sub-concept, can be used to construct a told subsumption hierarchy. Although such a hierarchy is usually incomplete and has many entailed subsumptions missing, it contains already known subsumptions indicating the closeness between concepts w.r.t. subsumption. Such a raw subsumption hierarchy can be represented as a directed graph with only one root, the $\top$ concept. A heuristic partitioning method can be defined by traversing the graph in a breadth-first way,

[1] This process does not show a full calling order of computing $A_4 \leq A_5$ for sake of brevity. For instance, $\top\_merge(A_7, A_6, \leq_\alpha, \leq_\beta)$ is not shown.

Algorithm 8. *build_partition*$(n, visited, G, P)$

```
input  : $n$: an concept name
         $visited$: a list recording visited concept names
         $G$: the syntax-based subsumption graph
         $P$: a concept names partition
output : $R$: a concept names partition
1  begin
2      $R \leftarrow \phi$;
3      $N \leftarrow get\_children(n, visited, G, P)$;
4      foreach $n' \in N$ do
5          if $n' \notin visited$ then
6              $P \leftarrow P \cup \{n'\}$;
7              $visited \leftarrow visited \cup \{n'\}$;
8              $build\_partition(n', visited, G, P)$;
9          end if
10     end foreach
11     $R \leftarrow P$;
12     return $R$;
13 end
```

starting from $\top$, and collecting traversed concepts into partitions. Algorithms 7 and 8 address this procedure.

4 Evaluation

Our experimental results clearly show the potential of merge-classification. We could achieve speedups up to a factor of 4 by using a maximum of 8 parallel workers, depending on the particular benchmark ontology. This speedup is in the range of what we expected and comparable to other reported approaches, e.g., the experiments reported for the ELK reasoner [16,17] also show speedups up to a factor of 4 when using 8 workers, although a specialized polynomial procedure is used for $\mathcal{EL}+$ reasoning that seems to be more amenable to concurrent processing than standard tableau methods.

We have designed and implemented a concurrent version of the algorithm so far. Our program[2] is implemented on the basis of the well-known reasoner JFact,[3] which is open-source and implemented in Java. We modified JFact such that we can execute a set of JFact reasoning kernels in parallel in order to perform the merge-classification computation. We try to examine the effectiveness of the merge-classification algorithm by adapting such a mature DL reasoner.

4.1 Experiment

A multi-processor computer, which has 4 octa-core processors and 128G memory installed, was employed to test the program. The Linux OS and 64-bit OpenJDK

[2] http://github.com/kejia/mc
[3] http://jfact.sourceforge.net

Table 1. Metrics of the test cases.

Ontology	Expressivity	Concept count	Axiom count
adult_mouse_anatomy	$\mathcal{ALE}+$	2753	9372
amphibian_gross_anatomy	$\mathcal{ALE}+$	701	2626
c_elegans_phenotype	$\mathcal{ALEH}+$	1935	6170
cereal_plant_trait	$\mathcal{ALEH}$	1051	3349
emap	$\mathcal{ALE}$	13731	27462
environmental_entity_logical_definitions	$\mathcal{SH}$	1779	5803
envo	$\mathcal{ALEH}+$	1231	2660
fly_anatomy	$\mathcal{ALEI}+$	6222	33162
human_developmental_anatomy	$\mathcal{ALEH}$	8341	33345
medaka_anatomy_development	$\mathcal{ALE}$	4361	9081
mpath	$\mathcal{ALEH}+$	718	4315
nif-cell	$\mathcal{S}$	376	3492
sequence_types_and_features	$\mathcal{SH}$	1952	6620
teleost_anatomy	$\mathcal{ALER}+$	3036	11827
zfa	$\mathcal{ALEH}+$	2755	33024

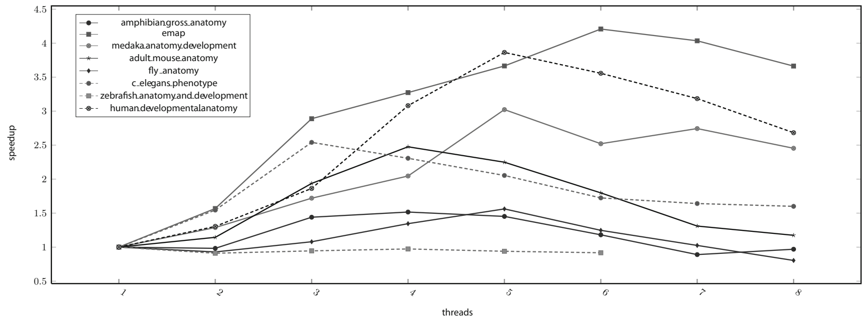


Fig. 5. The performance of parallelized merge-classification—I.

6 was employed in the tests. The JVM was allocated at least 16G memory initially, given that at most 64G physical memory was accessible. Most of the test cases were chosen from *OWL Reasoner Evaluation Workshop 2012 (ORE 2012)* data sets. Table 1 shows the test cases' metrics.

Each test case ontology was classified with the same setting except for an increased number of workers. Each worker is mapped to an OS thread, as indicated by the Java specification. Figure 5 and 6 show the test results.

In our initial implementation, we used an *even-partitioning* scheme. That is to say concept names are randomly assigned to a set of partitions. For the majority of the above-mentioned test cases we observed a small performance improvement below a speedup factor of 1.4, for a few an improvement of up to 4, and for others only a decrease in performance. Only overhead was shown in these test cases.

As mentioned in Sect. 3, we assumed that a heuristic partitioning might promote a better reasoning performance, e.g., a partitioning scheme considering subsumption axioms. This idea is addressed by Algorithms 7 and 8.

We implemented Algorithms 7 and 8 and tested the program. Our assumption has been proved by the test: Heuristic partitioning may improve reasoning performance where blind partitioning can not.

4.2 Discussion

Our experiment shows that with a heuristic divide scheme the merge-classification algorithm can increase reasoning performance. However, such performance promotion is not always tangible. In a few cases, the parallelized merge-classification merely degrades reasoning performance. The actual divide phase of our algorithm can influence the performance by creating better or worse partitions.

A heuristic divide scheme may result in a better performance than a blind one. According to our experience, when the division of the concepts from the domain is basically random, sometimes divisions contribute to promoting reasoning performance, while sometimes they do not. A promising heuristic divide scheme seems to be in grouping a family of concepts, which have potential subsumption relationships, into the same partition. Evidently, due to the presence of non-obvious subsumptions, it is hard to guess how to achieve such a good partitioning. We tried to make use of obvious subsumptions in axioms to partition closely related concepts into the same group. The tests demonstrate a clear performance improvement in a number of cases.

While in many cases merge-classification can improve reasoning performance, for some test cases its practical effectiveness is not yet convincing. We are still investigating the factors that influence the reasoning performance for these cases but cannot give a clear answer yet. The cause may be the large number of *general concept inclusion (GCI)* axioms of ontologies. Even with some more refined divide scheme, those GCI axioms can cause inter-dependencies between partitions, and may cause in the merge phase an increased number of subsumption tests. Also, the indeterminism of the merging schedule, i.e., the unpredictable

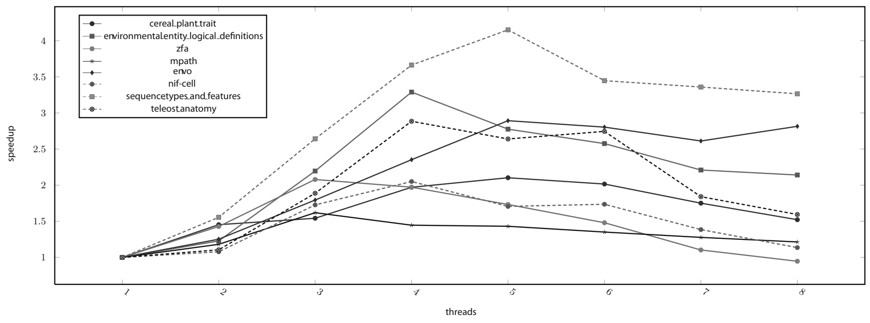


Fig. 6. The performance of parallelized merge-classification—II.

order of merging divides, needs to be effectively solved in the implementation, and racing conditions between merging workers as well as the introduced overhead may decrease the performance. In addition, the limited performance is caused by the experimental environment: Compared with a single chip architecture, the 4-chip-distribution of the 32 processors requires extra computational overhead, and the memory and thread management of JVM may decrease the performance of our program.

5 Related Work

A key functionality of a DL reasoning system is *classification*, computing all entailed subsumption relationships among named concepts. The generic top-search and bottom-search algorithms were introduced by [19] and extended by [2]. The algorithm is used as the standard technique for incrementally creating subsumption hierarchies of DL ontologies. Reference [2] also presented some basic traversal optimizations. After that, a number of optimization techniques have been explored [8,9,26]. Most of the optimizations are based on making use of the partial transitivity information in searching. However, research on how to use concurrent computing for optimizing DL reasoning has started only recently.

The merge-classification algorithm is suitable for concurrent computation implementation, including both shared-memory parallelization and (shared-memory or non-shared-memory) distributed systems. Several concurrency-oriented DL reasoning schemes have been researched recently. Reference [18] reported on experiments with a parallel $\mathcal{SHN}$ reasoner. This reasoner could process *disjunction* and *at-most cardinality restriction* rules in parallel, as well as some primary DL tableau optimization techniques. Reference [1] presented the first algorithms on parallelizing TBox classification using a shared global subsumption hierarchy, and the experimental results promise the feasibility of parallelized DL reasoning. References [16,17] reported on the ELK reasoner, which can classify $\mathcal{EL}$ ontologies *concurrently*, and its speed in reasoning about $\mathcal{EL}+$ ontologies is impressive. References [28,29] studied a parallel DL reasoning system. References [20,21] proposed the idea of applying a constraint programming solver. Besides the shared-memory concurrent reasoning research mentioned above, non-shared-memory distributed concurrent reasoning has been investigated recently by [22,25].

Merge-classification needs to divide ontologies. Ontology partitioning can be considered as a sort of *clustering* problem. These problems have been extensively investigated in networks research, such as [6,7,30]. Algorithms adopting more complicated heuristics in the area of ontology partitioning, have been presented in [5,10–12,14].

Our merge-classification approach employs the well-known *divide and conquer* strategy. There is sufficient evidence that these types of algorithms are well suited to be processed in parallel [4,15,27]. Some experimental works about parallelized merge sort are reported in [23,24].

6 Conclusion

The approach presented in this paper has been motivated by the observation that (i) multi-processor/core hardware is becoming ubiquitously available but standard OWL reasoners do not yet make use of these available resources; (ii) Although most OWL reasoners have been highly optimized and impressive speed improvements have been reported for reasoning in the three tractable OWL profiles, there exist a multitude of OWL ontologies that are outside of the three tractable profiles and require long processing times even for highly optimized OWL reasoners. Recently, concurrent computing has emerged as a possible solution for achieving a better scalability in general and especially for such difficult ontologies, and we consider the research presented in this paper as an important step in designing adequate OWL reasoning architectures that are based on concurrent computing.

One of the most important obstacles in successfully applying concurrent computing is the management of overhead caused by concurrency. An important factor is that the load introduced by using concurrent computing in DL reasoning is usually remarkable. Concurrent algorithms that cause only a small overhead seem to be the key to successfully apply concurrent computing to DL reasoning.

Our merge-classification algorithm uses a *divide and conquer* scheme, which is potentially suitable for low overhead concurrent computing since it rarely requires communication among divisions. Although the empirical tests show that the merge-classification algorithm does not always improve reasoning performance to a great extent, they let us be confident that further research is promising. For example, investigating what factors impact the effectiveness and efficiency of the merge-classification may help us improve the performance of the algorithm further.

At present our work adopts a heuristic *partitioning* scheme at the divide phase. Different divide schemes may produce different reasoning performances. We are planning to investigate better divide methods. Furthermore, our work has only researched the performance of the *concurrent* merge-classification so far. How the number of division impacts the reasoning performance in a single thread and a multiple threads setting needs be investigated in more detail.

Acknowledgements. We are grateful for Ralf Möller at Hamburg University of Technology in giving us access to their equipment that was used to conduct the presented evaluation.

References

1. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics–first experimental results. In: Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 485–490 (2010)
2. Baader, F., Hollunder, B., Nebel, B., Profitlich, H.J., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. Appl. Intell. **4**(2), 109–132 (1994)

3. Baader, F., et al.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, New York (2003)
4. Cole, R.: Parallel merge sort. SIAM J. Comput. **17**(4), 770–785 (1988)
5. Doran, P., Tamma, V., Iannone, L.: Ontology module extraction for ontology reuse: an ontology engineering perspective. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, pp. 61–70 (2007)
6. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996)
7. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. Nat. Acad. Sci. U.S.A. **99**(12), 7821–7826 (2002)
8. Glimm, B., Horrocks, I., Motik, B., Stoilos, G.: Optimising ontology classification. 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. 225–240. Springer, Heidelberg (2010)
9. Glimm, B., Horrocks, I., Motik, B., Shearer, R., Stoilos, G.: A novel approach to ontology classification. Web Semantics: Science, Services and Agents on the World Wide Web **14**, 84–101 (2012)
10. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: A logical framework for modularity of ontologies. In: Proceedings International Joint Conference on Artificial Intelligence, pp. 298–304 (2007)
11. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: theory and practice. J. Artif. Intell. Res. **31**(1), 273–318 (2008)
12. Grau, B.C., Parsia, B., Sirin, E., Kalyanpur, A.: Modularizing OWL ontologies. In: K-CAP 2005 Workshop on Ontology Management (2005)
13. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC, London (2009)
14. Hu, W., Qu, Y., Cheng, G.: Matching large ontologies: a divide-and-conquer approach. Data Knowl. Eng. **67**(1), 140–160 (2008)
15. Jeon, M., Kim, D.: Parallel merge sort with load balancing. Int. J. Parallel Prog. **31**(1), 21–33 (2003)
16. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of $\mathcal{EL}$ ontologies. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 305–320. Springer, Heidelberg (2011)
17. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: from polynomial procedures to efficient reasoning with EL ontologies (2013) (submitted to a journal)
18. Liebig, T., Müller, F.: Parallelizing tableaux-based description logic reasoning. In: Meersman, R., Tari, Z. (eds.) OTM-WS 2007, Part II. LNCS, vol. 4806, pp. 1135–1144. Springer, Heidelberg (2007)
19. Lipkis, T.: A KL-ONE classifier. In: Proceedings of the 1981 KL-ONE Workshop, pp. 128–145 (1982)
20. Meissner, A.: A simple parallel reasoning system for the $\mathcal{ALC}$ description logic. In: Nguyen, N.T., Kowalczyk, R., Chen, S.-M. (eds.) ICCCI 2009. LNCS, vol. 5796, pp. 413–424. Springer, Heidelberg (2009)
21. Meissner, A., Brzykcy, G.: A parallel deduction for description logics with ALC language. Knowl.-Driven Comput. **102**, 149–164 (2008)
22. Mutharaju, R., Hitzler, P., Mateti, P.: DistEL: A distributed EL+ ontology classifier. In: Proceedings of The 9th International Workshop on Scalable Semantic Web Knowledge Base Systems (2013)

23. Radenski, A.: Shared memory, message passing, and hybrid merge sorts for standalone and clustered SMPs. In: The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, vol. 11, pp. 367–373 (2011)
24. Rolfe, T.J.: A specimen of parallel programming: parallel merge sort implementation. ACM Inroads **1**(4), 72–79 (2010)
25. Schlicht, A., Stuckenschmidt, H.: Distributed resolution for ALC - first results. In: Proceedings of the Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (2008)
26. Shearer, R., Horrocks, I., Motik, B.: Exploiting partial information in taxonomy construction. In: Grau, B.G., Horrocks, I., Motik, B., Sattler, U. (eds.) Proceedings of the 2009 International Workshop on Description Logics. CEUR Workshop Proceedings, Oxford, UK, vol. 477, 27–30 July 2009
27. Todd, S.: Algorithm and hardware for a merge sort using multiple processors. IBM J. Res. Dev. **22**(5), 509–517 (1978)
28. Wu, K., Haarslev, V.: A parallel reasoner for the description logic ALC. In: Proceedings of the 2012 International Workshop on Description Logics (2012)
29. Wu, K., Haarslev, V.: Exploring parallelization of conjunctive branches in tableau-based description logic reasoning. In: Proceedings of the 2013 International Workshop on Description Logics (2013)
30. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: SCAN: a structural clustering algorithm for networks. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 824–833 (2007)