ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010

doi:10.3233/978-1-60750-606-5-485

Parallel TBox Classification in Description Logics – First Experimental Results

Mina Aslani and **Volker Haarslev**[1]

Abstract. One of the most frequently used inference services of description logic reasoners classifies all named classes of OWL ontologies into a subsumption hierarchy. Due to emerging OWL ontologies from the web community consisting of up to hundreds of thousand of named classes and the increasing availability of multi-processor and multi- or many-core computers, we extend our work on parallel TBox classification and propose a new algorithm that is sound and complete and demonstrates in a first experimental evaluation a low overhead w.r.t. subsumption tests (less than 3%) if compared with sequential classification.

1 Motivation

Due to the recent popularity of OWL ontologies in the web one can observe a trend toward the development of very large or huge OWL-DL ontologies. For instance, well known examples from the bioinformatics or medical community are SNOMED, UMLS, GALEN, or FMA. Some (versions) of these ontologies consist of more than hundreds of thousands of named concepts/classes and have become challenging even for the most advanced and optimized description logic (DL) reasoners. Although specialized DL reasoners for certain sublogics (e.g., CEL for EL++) and OWL-DL reasoners such as FaCT++, Pellet, HermiT, or RacerPro could demonstrate impressive speed enhancements due to newly designed optimization techniques, one can expect the need for parallelizing description logic inference services in the near future in order to achieve a web-like scalability where we have to consider millions of concepts or hundreds of thousands of concepts that cause very difficult subsumptions tests. Our research is also strongly motivated by recent trends in computer hardware where processors feature multi-cores (2 to 8 cores) or many-cores (tens or even hundreds of cores). These processors promise significant speed-ups for algorithms exploiting so-called thread-level parallelism. This type of parallelism is very promising for DL reasoning algorithms that can be executed in parallel but might share common data structures (e.g., and/or parallelism in proofs, classification of TBoxes, ABox realization or query answering).

First approaches on scalable reasoning algorithms for ABoxes (sets of declarations about individuals) were investigated with the Racer architecture [11] where novel instance retrieval algorithms were developed and analyzed, which exploit a variety of techniques such as index maintenance, dependency analysis, precompletion generation, etc. Other research focused on scalable ABox reasoning with optimization techniques to partition ABoxes into independent parts and/or creating condensed (summary) ABoxes [8, 9, 6]. These approaches rely on the observation that the structure of particular ABoxes is often redundant and these ABoxes contain assertions not needed for ABox consistency checking or query answering.

Parallel algorithms for description logic reasoning were first explored in the FLEX system [3] where various distributed message-passing schemes for rule execution were evaluated. The reported results seemed to be promising but the research suffered from severe limitations due to the hardware available for experiments at that time. The only other approach on parallelizing description logic reasoning [13] reported promising results using multi-core/processor hardware, where the parallel treatment of disjunctions and individual merging (due to number restrictions) is explored. In [14] an approach on distributed reasoning for $\mathcal{ALCHIQ}$ is presented that is based on resolution techniques but does not address optimizations for TBox (set of axioms) classification. There also exists work on parallel distributed RDF inferencing (e.g., [17]) and parallel reasoning in first-order theorem proving but due to completely different proof techniques (resolution versus tableaux) and reasoning architectures this is not considered as relevant here.

There has also been substantial work on reasoning through modularity and partitioning knowledge bases (e.g., [7, 5, 4]). In [7], the proposed greedy algorithm performs automated partitioning, and the authors have investigated how to reason effectively with partitioned sets of logical axioms that have overlap in content and may even require different reasoning engines. Their partition-based reasoning algorithms have been proposed for reasoning with logical theories in propositional and first-order predicate logic that are decomposed into related partitions of axioms. In [5], a logic-based framework for modularity of ontologies is proposed. This formalization is very interesting for ontologies that can be modularized. For these cases, every module could be assigned to a particular thread and classified in parallel. The approach reported in [4] also proposed a technique for incremental ontology reasoning that reuses the results obtained from previous computations. This technique is based on the notion of a module and can be applied to arbitrary queries against ontologies expressed in OWL-DL. The approach focused on a particular kind of modules that exhibit a set of compelling properties and apply their method to incremental classification of OWL-DL ontologies. The techniques do not depend on a particular reasoner or reasoning method and can be easily implemented in any existing prover.

In the following we extend our work on parallel TBox classification [1] and propose a new algorithm that is sound *and* complete although it runs in parallel. Our first approach [1] did not ensure completeness. The implemented prototype system performs parallel TBox classification with various parameters such as number of threads, size of partitions assigned to threads, etc. First results from a preliminary evaluation look very promising and indicate a very low overhead.

[1] Concordia University, Montreal, Canada,
email: {m_aslani,haarslev}@cse.concordia.ca

2 The New Parallel TBox Classifier

This section describes the architecture of the implemented system and its underlying *sound and complete* algorithm for parallel classification of DL ontologies. To compute the hierarchy in parallel, we developed a multi-threaded architecture providing control parameters such as number of threads, number of concepts (also called partition size) to be inserted per thread, and strategies used to partition a given set of concepts. Our system reads an input file containing a list of concept names to be classified and information about them. The per-concept information available in the file includes its name, parents (in the complete taxonomy), told subsumers, told disjoints, and pseudo model information. The information about parents is used to compute the set of ancestors and descendants of a concept. Told information consists of subsumers and disjoints that can be easily extracted from axioms without requiring proof procedures, e.g. the axiom $A \sqsubseteq B \sqcap \neg C$ would result in information asserting B as told subsumer of A and C as told disjoint of A. With the exception of told subsumers this information is only used for (i) emulating a tableau subsumption test, i.e., by checking whether a possible subsumer (subsumee) is in the list of ancestors (descendants) of a given concept, and (ii) in order to verify the completeness of the taxonomy computed by the parallel classifier. The input information substitutes for an implemented tableaux reasoning procedure, hence makes the parallel classifier independent of a particular DL logic or reasoner. Currently, RacerPro[2] is used to generate this file for a given OWL-DL ontology after performing TBox classification.

The told subsumer information is passed to a preprocessing algorithm which creates a taxonomy skeleton based on the already known (told) subsumptions and generates a topological-order list (e.g. depth-first traversal). Using a topological sorting algorithm, the partial order can be serialized such that a total order between concept names (or sets of concept names) is defined. During classification, the concept names are processed according to their topological order. In our topological order list, from left to right, parent concepts precede child concepts.

To manage concurrency in our system, at least two shared-memory approaches could be taken into account by using either (i) sets of local trees (so-called ParTree approach) or (ii) one global tree. In the ParTree algorithm [15] a local tree would be assigned to each thread, and after all the threads have finished the construction of their local hierarchy, the local trees need to be merged into one global tree. TBox classification through a local tree algorithm would not need any communication or synchronization between the threads. ParTree is well suited for distributed systems which do not have shared memory. The global tree approach was chosen because it implements a shared space which is accessible to different threads running in parallel and avoids the large scale overhead of ParTree on synchronizing local trees. To ensure data integrity a lock mechanism for single nodes is used. This allows a proper lock granularity and helps to increase the number of simultaneous write accesses to the subsumption hierarchy under construction.

Most TBox classification algorithms are based on two (symmetric) tasks (e.g., see [2]). The first phase (top search) determines the parents of a given concept to be inserted into the subsumption tree. It starts with the top concept ($\top$) and tries to push the given concept below the children of the current concept and repeats this process with the goal to push the given concept as much to the bottom of the subsumption tree as possible. Whenever a concept in the tree subsumes the given concept, it is pushed below this subsumer. The second phase (bottom search) determines the children of a given concept. It starts with the bottom concept ($\bot$) and tries to move the given concept above the parents of the current concept and repeats this process with the goal to move the current concept up in the tree as much as possible. Whenever a concept in the tree is subsumed by the given concept, it is moved above of this subsumee. Eventually, the given concept is correctly positioned in the current subsumption hierarchy. Both phases tag nodes of the tree ('visited', 'positive', 'negative') to prune the search and avoid visiting already processed nodes. For instance, 'positive' is used to tag nodes already known as (told) subsumers and 'negative' for already known as (told) disjoints.

The work in [2] is an example for algorithms that incrementally construct a subsumption tree and are highly optimized for sequential execution. In [10] some of these techniques were extended to better deal with huge TBox hierarchies but these algorithms are still based on a sequential execution. A recent approach [16] on TBox classification exploits partial information about OWL subclass relationships to reduce the number of subsumption tests and, thus, improves the algorithms presented in [2].

[2] http://racer-systems.com

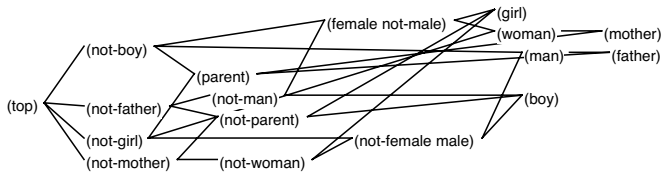

Figure 1. Complete subsumption hierarchy for yaya-1

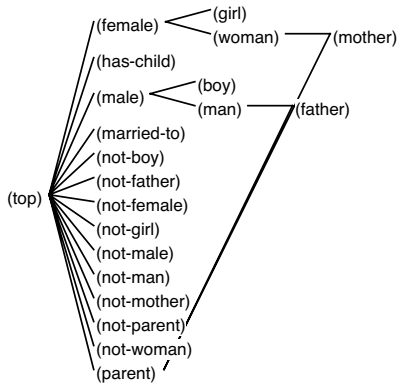

Figure 2. Told subsumer hierarchy for yaya-1

2.1 Example Scenario

In [1] the degree of incompleteness caused by classifying partitions of concepts in parallel was tested. For a variety of ontologies it turned out that a surprisingly few number of subsumptions were missed. This motivated the work in this paper. In the following we illustrate the only two scenarios which may cause that a concept is misplaced in the taxonomy due to parallel classification. For sake of brevity we use a very small ontology named yaya-1 with 16 concepts (see Fig. 1 and 2).

For this example, we configured our system so that it runs with 4

thread#1 ⟶ (female not-male), girl, parent
thread#2 ⟶ woman, mother, (male not-female)
thread#3 ⟶ man, boy, father
thread#4 ⟶ not-boy, not-father, not-girl
thread#1 ⟶ not-man, not-mother, not-parent, not-woman

Figure 3. Concept assignments to each thread for classifying yaya-1

Algorithm 1 parallel_tbox_classification(*concept_list*)

topological_order_list ← topological_order(*concept_list*)
repeat
  wait until an idle thread t_i becomes available
  select a partition p_i from *topological_order_list*
  run thread t_i with insert_partition(p_i, t_i)
until all concepts in *topological_order_list* are inserted

threads and 3 number-of-tasks-per-thread. As explained previously, in parallel classification the topological sort order divides concept partitions between the threads (e.g. round-robin). For instance, in Fig. 3 a list of concepts allocated to each thread is shown. The only two possible scenarios (illustrated in the proof of Proposition 1 below) that may lead to a situation where the correct place of a concept in the hierarchy is overlooked are described as follows.

Scenario I: In top search, as the new concept is being pushed downward, right after the children of the current concept have been processed, at least one new child is added by another thread. In this scenario, the top search for the new concept is not aware of the recent change and this might cause missing subsumptions if there is any interaction between the new concept and the added children. The same might happen in bottom search if the bottom search for the new concept is not informed of the recent change to the list of parents of the current node.

Scenario II: Between the time that top and bottom search have been started to find the location of a new concept in the taxonomy and the time its location has been decided, a different thread has independently placed at least one concept into another (possibly disjoint) part of the hierarchy which the new concept has an interaction with. Again, this might cause missing subsumptions.

In our example (yaya-1), due to the small size of the taxonomy, scenario I was not encountered, however, scenario II occurred in our experiments because thread#1 inserted (female not-male)[3] and thread#2 added woman independently into the taxonomy and due to the parallelism each thread did not have any information regarding the latest concept insertion by other threads (see also Fig. 3). Hence, both (female not-male) and woman were initially placed directly under the top concept although woman should be a child of (female not-male) (see Fig. 1). This was discovered and corrected by executing lines 6-7, 16-17, and 25-37 in Algorithm 2.

2.2 Algorithms for Parallel Classification

The procedure parallel_tbox_classification is sketched in Algorithm 1. It is called with a list of named concepts and sorts them in topological order w.r.t. to the initial taxonomy created from the already known told ancestors and descendants of each concept (using the told subsumer information). The classifier assigns in a round-robin manner partitions with a fixed size from the concept list to idle threads and activates these threads with their assigned partition using the procedure insert_partition outlined in Algorithm 2. All threads work in parallel with the goal to construct a global subsumption tree (taxonomy). They also share a global array *inserted_concepts* indexed by thread identifications. Nodes in the global tree as well as entries in the array are locked for modification.

The procedure insert_partition inserts all concepts of a given partition into the global taxonomy. For inserting a concept or updating its parents or children, it locks the corresponding nodes. It first performs for each concept *new* the top-search phase (starting from the top concept) and possibly repeats the top-search phase for *new* if other threads updated the list of children of its parents and there is an interaction between *new* and the added children. Then, it sets the parents of *new* and adds *new* for each parent to its list of children.

Algorithm 2 insert_partition(*partition,id*)

```
 1: lock(inserted_concepts(id))
 2: inserted_concepts(id) ← ∅
 3: unlock(inserted_concepts(id))
 4: for all new ∈ partition do
 5:   parents ← top_search(new,⊤)
 6:   while ¬ consistent_in_top_search(parents,new) do
 7:     parents ← top_search(new,⊤)
 8:   lock(new)
 9:   predecessors(new) ← parents
10:   unlock(new)
11:   for all pred ∈ parents do
12:     lock(pred)
13:     successors(pred) ← successors(pred) ∪ {new}
14:     unlock(pred)
15:   children ← bottom_search(new,⊥)
16:   while ¬ consistent_in_bottom_search(children,new) do
17:     children ← bottom_search(new,⊥)
18:   lock(new)
19:   successors(new) ← children
20:   unlock(new)
21:   for all succ ∈ children do
22:     lock(succ)
23:     predecessors(succ) ← predecessors(succ) ∪ {new}
24:     unlock(succ)
25:   check ←
        check_if_concept_inserted(new, inserted_concepts(id))
26:   if check ≠ 0 then
27:     if check = 1 ∨ check = 3 then
28:       new_predecessors ← top_search(new,⊤)
29:       lock(new)
30:       predecessors(new) ← new_predecessors
31:       unlock(new)
32:     if check = 2 ∨ check = 3 then
33:       new_successors ← bottom_search(new,⊥)
34:       lock(new)
35:       successors(new) ← new_successors
36:       unlock(new)
37:   insert_concept(new,predecessors(new),successors(new))
38:   for all busy threads t_i ≠ id do
39:     lock(inserted_concepts(t_i))
40:     inserted_concepts(t_i) ← inserted_concepts(t_i) ∪ {new}
41:     unlock(inserted_concepts(t_i))
```

[3] This notation indicates that the concepts female and not-male are synonyms for each other.

Algorithm 3 top_search(*new*,*current*)

```
mark(current,'visited')
pos-succ ← ∅
captured_successors(new)(current) ← successors(current)
for all y ∈ successors(current) do
  if enhanced_top_subs(y,new) then
    pos-succ ← pos-succ ∪ {y}
if pos-succ = ∅ then
  return {current}
else
  result ← ∅
  for all y ∈ pos-succ do
    if y not marked as 'visited' then
      result ← result ∪ top_search(new,y)
  return result
```

Algorithm 4 enhanced_top_subs(*current*,*new*)

```
if current marked as 'positive' then
  return true
else if current marked as 'negative' then
  return false
else if for all z ∈ predecessors(current)
          enhanced_top_subs(z,new)
       and subsumes(current,new) then
  mark(current,'positive')
  return true
else
  mark(current,'negative')
  return false
```

Afterwards the bottom-search phase (starting from the bottom concept) is performed. Analogously to the top-search phase the bottom search is possibly repeated and sets the children of *new* and updates the parents of the children of *new*. After finishing the top and bottom search for *new* it is checked again whether other threads updated its entry in *inserted_concepts* and the top and/or bottom search needs to be repeated. Finally, *new* is properly inserted into the hierarchy by updating its parents and children accordingly (line 37 in Algorithm 2) and also added to the entries in *inserted_concepts* of all other busy threads.

In order to avoid unnecessary tree traversals and tableau subsumption tests when computing the subsumption hierarchy, the parallel classifier adapted the enhanced traversal method [2], which is an algorithm that was designed for sequential execution. Algorithm 3 and 4 outline the traversal procedures for the top-search phase.

The procedure top_search outlined in Algorithm 3 recursively traverses the taxonomy top-down from a current concept and tries to push the new concept down the taxonomy as far as possible by traversing the children of the current concept. It uses an auxiliary procedure enhanced_top_subs (outlined in Algorithm 4) which itself uses an auxiliary procedure subsumes (not specified here) that implements a subsumption test. The procedure enhanced_top_subs tests whether *current* subsumes *new*. If the node *current* was tagged these tags are used to prune the search, otherwise the parents of the node *current* are recursively traversed.

In a symmetric manner the procedure bottom_search traverses the taxonomy bottom-up from a current concept and tries to push the *new* concept up the taxonomy as far as possible. It uses an auxiliary procedure enhanced_bottom_subs. Both procedures are omitted for ease of presentation.

Algorithm 5 consistent_in_top_search(*parents*,*new*)

```
for all pred ∈ parents do
  if successors(pred) ≠ captured_successors(new)(pred) then
    diff ← successors(pred) \ captured_successors(new)(pred)
    for all child ∈ diff do
      if subsumption_possible(child,new) then
        return false
return true
```

Algorithm 6 check_if_concept_inserted(*new*,*inserted_concepts*)

```
if inserted_concepts = ∅ then
  return 0
else
  for all concept ∈ inserted_concepts do
    if subsumption_possible(concept,new) then
      if subsumption_possible(new,concept) then
        return 3
      else
        return 1
    else if subsumption_possible(new,concept) then
      if subsumption_possible(concept,new) then
        return 3
      else
        return 2
  return 0
```

To resolve the possible incompleteness caused by parallel classification, we utilize Algorithms 5 and 6. The procedure consistent_in_bottom_search is not shown here because it mirrors consistent_in_top_search.

Algorithms 5 illustrate the solution for scenario I described in Section 2.1. As already described, in top search we start traversing from the top concept to locate the concept *new* in the taxonomy. At time *t1*, when top_search is called, we capture the children information "captured_successors" of the concept *current*; the children information is stored relative[4] to the concept *new* being inserted (we use an array of arrays) and captures the successors of the concept *current* (see Algorithm 3). As soon as top_search is finished at time *t2*, and the parents of the concept *new* have been determined, we check if there has been any update on the children list of the computed parents for *new* between *t1* and *t2* (e.g., see Algorithm 5 on how this is discovered). If there is any inconsistency and also if there is a subsumption possible[5] between *new* and any concept newly added to the children list, we rerun top_search until there is no inconsistency (see line 6 in Algorithm 2).

The same process as illustrated in Algorithm 5 happens in bottom search. The only difference is that parents information is captured when bottom search starts; and when bottom search finishes, the inconsistency and interaction is checked between the parent list of the computed children for *new* and the "captured_predecessors".

Algorithm 6 describes the solution for scenario II; every time a thread inserts a concept in the taxonomy, it notifies the other threads by adding the concept name to their "inserted_concepts" list. Therefore, as soon as a thread finds the parents and children of the *new* concept by running top_search and bottom_search; it checks if there is

[4] Otherwise a different thread could overwrite captured_successors for node *current*. This is now prevented because each concept (*new*) is inserted by only one thread.

[5] This is checked by subsumption_possible using pseudo model merging [12], where a sound but incomplete test for non-subsumption on cached pseudo models of named concepts and their negation is utilized.

any interaction between *new* concept and the concepts located in the "inserted_concepts" list. Based on the interaction, top_search and/or bottom_search need to be repeated accordingly.

Proposition 1 (Completeness of Parallel TBox Classifier) *The proposed algorithms are complete for TBox classification.*

TBox classification based on top search and bottom search is complete in the sequential case. This means that the subsumption algorithms will find all subsumption relationships between concepts of a partition assigned to a single thread. The threads lock and unlock nodes whenever they are updating the information about a node in the global subsumption tree. Thus, we need to consider only the scenarios where two concepts C and D are inserted in parallel by different threads (e.g., thread#1 inserts concept C while thread#2 inserts concept D). In principle, if top (bottom) search pushed a new concept down (up), the information about children (parents) of a traversed node E could be incomplete because another thread might later add more nodes to the parents or children of E that were not considered when determining whether the concept being inserted subsumes or is subsumed by any of these newly added nodes. This leads to two scenarios that need to be examined for incompleteness.

W.l.o.g. we restrict our analysis to the case where a concept C is a parent of a concept D in the complete subsumption tree (CT). Let us assume that our algorithms would not determine this subsumption, i.e., in the computed (incomplete) tree (IT) the concept C is not a parent of D.

Case I: top_search incomplete for D: After D has been pushed down the tree IT as far as possible by top search (executed by thread#2) and top search has traversed the children of a concept E and E has become the parent of D, C is inserted by thread#1 as a new child of E. In line 6 of Algorithm 2 top_search is iteratively repeated for the concept *new* as long as consistent_in_top_search finds a discrepancy between the captured and current successors of the parents of the newly inserted concept *new*. After finishing top and bottom search, Algorithm 2 checks again in lines 27-28 whether top search needs to be repeated due to newly added nodes. If any of the newly added children of D would subsume C and become a parent of C, the repeated execution of top_search would find this subsumption. This contradicts our assumption.

Case II: bottom_search incomplete for C: After C has been pushed up the tree IT as far as possible by bottom search (executed by thread#1) and bottom search has traversed the parents of a concept E and E has become a child of C, D is inserted by thread#2 as a new parent of E. In line 16 of Algorithm 2 bottom_search is iteratively repeated for the concept *new* as long as consistent_in_bottom_search finds a discrepancy between the captured and current predecessors of the children of the newly inserted concept *new*. After finishing top and bottom search, Algorithm 2 checks again in lines 32-33 whether bottom search needs to be repeated due to newly added nodes. If C would subsume any of the newly added parents of D and it would become a child of C, the repeated execution of bottom_search would find this subsumption. This contradicts our assumption.

3 Evaluation

The Parallel TBox Classifier has been developed to speed up the classification time especially for large ontologies by utilizing parallel threads sharing the same memory. The benchmarking can be configured so that it runs various experiments over ontologies. We evaluated it with a collection of 8 mostly publicly available ontologies. Their name, size in number of named concepts, and used DL is shown in Table 1. As mentioned in the previous section, two parameters influence the parallel TBox classification, namely number of tasks/concepts per thread and number of threads; the number of tasks/concepts per thread was set to 5 and number of threads to 2 in our empirical experiments. This evaluation focussed only on the number of performed subsumption tests. The runtime of our system and the runtime of RacerPro were not considered yet.

Table 1. Characteristics of the used test ontologies.

Ontology name	DL language	No. of named concepts
Embassi-2	$\mathcal{ALCHN}$	657
Embassi-3	$\mathcal{ALCHN}$	1,121
Galen	$\mathcal{SHN}$	2,730
Galen1	$\mathcal{ALCH}$	2,730
Galen2	$\mathcal{ELH}$	3,928
FungalWeb	$\mathcal{ALCHIN}(\mathcal{D})$	3,603
Umls-2	$\mathcal{ALCHIN}(\mathcal{D})$	9,479
Tambis-2a	$\mathcal{ELH}$	10,116

To better compare the performance between the sequential and parallel case, we assume that every subsumption test runs in time *t1* and in the sequential and parallel case the same amount of time is used for an executed subsumption test. Subsumption tests can be expensive and, hence, are preferred to be avoided by optimization techniques such as pseudo model merging [12]. The ratio illustrated in Equation 1 uses $TotSubsTests_s$, the number of times a subsumption test was computed in the sequential case, and *MaxOfSubTestsInEachThread*, the maximum number of subsumption tests performed in each threads. Similarly, Equation 2 defines the overhead (where the index p refers to the parallel case).

$$Ratio = \frac{MaxOfSubTestsInEachThread}{TotSubsTests_s} \quad (1)$$

$$Overhead = \frac{TotSubsTests_p - TotSubsTests_s}{TotSubsTests_s} \quad (2)$$

Table 2 shows an excellent performance increase and a surprisingly small overhead when using the Parallel TBox Classifier. Using two threads the maximum of number of subsumption test for all ontologies could be reduced to roughly one half compared to the sequential case. The overhead as defined in Equation 2 varies between 0.13% and 2.62%. The overhead is mostly determined by the quality of the told subsumers and disjoints information, the imposed order of traversal within a partitioning, and the division of the ordered concept list into partitions. In general, one tries to insert nodes as close as possible to their final order in the tree using a top to bottom strategy.

Figure 4 shows five graphs depicting the structure of selected subsumption hierarchies, where the length of a line reflects the number of nodes on this level (shown from top to bottom). As can be observed in Figure 4, the shapes of the subsumption hierarchies are quite different and the order of inserting concepts affects the number of top and bottom searches. In the sequential case usually a topological sort-order based on told subsumers and disjoints information is a good approximation to minimize the number of bottom searches. In the parallel case, a topological sort-order is also used but its effectiveness is affected by the way such a sorted list is partitioned. Ideally, partitions should resemble small subtrees that interact with other subtrees as little as possible. Obviously, different shapes of subsumption trees resulting in different sets of partitions can affect the overhead in the parallel case. In our current evaluation we did not test different partitioning schemes yet and used an uninformed round-robin scheme to divide the sorted list into partitions.

Table 2. Subsumptions tests and their ratio for the test ontologies.

	Embassi-2	**Embassi-3**	**Galen**	**Galen1**	**Galen2**	**FungalWeb**	**Umls-2**	**Tambis-2a**
Subs. Tests in sequent.	154,034	420,912	2,706,412	2,688,107	5,734,976	4,996,932	87,423,341	36,555,225
Subs. Tests in thread#1	76,267	217,324	1,363,321	1,367,302	2,929,276	2,518,676	44,042,203	18,342,944
Subs. Tests in thread#2	77,767	214,633	1,354,297	1,348,281	2,893,716	2,490,329	44,025,988	18,261,532
Worst Case Ratio	50.48%	51.63%	50.37%	50.86%	51.07%	50.40%	50.37%	50.17%
Overhead	1.64%	2.62%	0.41%	1.02%	1.53%	0.24%	0.73%	0.13%

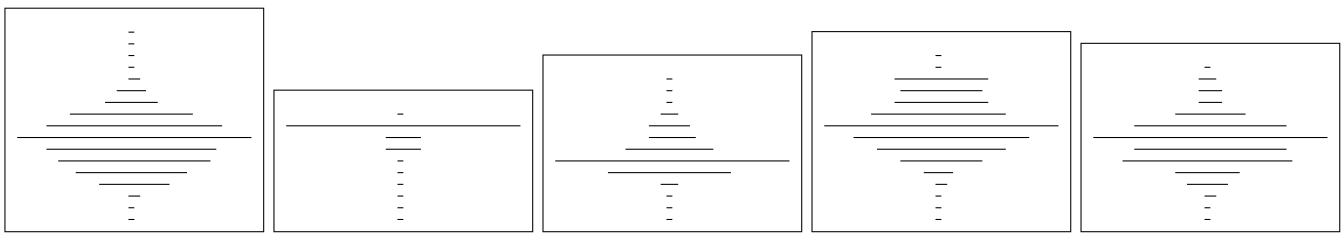

Figure 4. Subsumption tree structure of Galen, Umls-2, FungalWeb, Tambis-2a, and Embassi-3

4 Conclusion

In this paper, we described an architecture for parallelizing well-known algorithms for TBox classification. Our work is targeted for ontologies where independent partitions cannot be easily constructed; therefore we did not use the previously mentioned approaches in our system. The first experimental evaluation of our techniques shows that the results are very promising because the overhead for ensuring completeness is surprisingly small. In our next steps we plan to extend our tests with different configurations of threads and partition sizes and a larger variety of test ontologies. We intend to feed recorded runtimes for performing single subsumption tests into our system in order to make the computation of the overhead more accurate. We also plan to implement and test our approach in a multi-core and multi-processor environment.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their helpful comments.

REFERENCES

[1] M. Aslani and V. Haarslev, 'Towards parallel classifcation of TBoxes', in *Proceedings of the 2008 International Workshop on Description Logics (DL-2008), Dresden, Germany, May 13-16*, (2008).

[2] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.J. Profitlich, 'An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on', *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, **4**(2), 109–132, (1994).

[3] F. Bergmann and J. Quantz, 'Parallelizing description logics', in *Proc. of 19th Ann. German Conf. on Artificial Intelligence*, LNCS, pp. 137–148. Springer-Verlag, (1995).

[4] B. Cuenca Grau, C. Halaschek-Wiener, and Y. Kazakov, 'History matters: Incremental ontology reasoning using modules', in *Proc. of the 6th Int. Semantic Web Conf. (ISWC 2007), Busan, South Korea, Nov. 11-15*, (2007).

[5] B. Cuenca Grau, I. Horrocks, Y. Kazakov, and U. Sattler, 'A logical framework for modularity of ontologies', in *In Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007), Busan, South Korea, Nov. 11-15*, pp. 298–303, (2007).

[6] J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, L. Ma, E. Schonberg, and K. Srinivas, 'Scalable semantic retrieval through summarization and refinement', in *21st Conf. on Artificial Intelligence (AAAI)*, pp. 299–304. AAAI Press, (2007).

[7] A. Eyal and S. Mcllraith, 'Partition-based logical reasoning for first-order and propositional theories', *Artifical Intelligence*, **162**(1-2), 49–88, (2005).

[8] A. Fokoue, A. Kershenbaum, L. Ma, E. Schonberg, and K. Srinivas, 'The summary Abox: Cutting ontologies down to size', in *Proc. of Int. Semantic Web Conf. (ISWC)*, volume 4273 of *LNCS*, pp. 343–356. Springer-Verlag, (2006).

[9] Y. Guo and J. Heflin, 'A scalable approach for partitioning OWL knowledge bases', in *Proc. 2nd Int. Workshop on Scalable Semantic Web Knowledge Base Systems, Athens, USA*, pp. 47–60, (2006).

[10] V. Haarslev and R. Möller, 'High performance reasoning with very large knowledge bases: A practical case study', in *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence, IJCAI-01, Aug. 4-10, Seattle, USA*, pp. 161–166. Morgan Kaufmann, (2001).

[11] V. Haarslev and R. Möller, 'On the scalability of description logic instance retrieval', *Journal of Automated Reasoning*, **41**(2), 99–142, (2008).

[12] V. Haarslev, R. Möller, and A.-Y. Turhan, 'Exploiting pseudo models for TBox and ABox reasoning in expressive description logics', in *Proc. of the Int. Joint Conf. on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy*, LNCS, pp. 61–75, (June 2001).

[13] T. Liebig and F. Müller, 'Parallelizing tableaux-based description logic reasoning', in *Proc. of 3rd Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS '07), Vilamoura, Portugal, Nov 27*, volume 4806 of *LNCS*, pp. 1135–1144. Springer-Verlag, (2007).

[14] A. Schlicht and H. Stuckenschmidt, 'Distributed resolution for expressive ontology networks', in *Web Reasoning and Rule Systems, 3rd Int. Conf. (RR 2009), Chantilly, VA, USA, Oct. 25-26, 2009*, pp. 87–101, (2009).

[15] H. Shan and J. P. Singh, 'Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance', in *12th Int. Parallel Processing Symposium (IPPS '98), March 30 - April 3, 1998, Orlando, Florida, USA*, pp. 475–484, (1998).

[16] R. Shearer and I. Horrocks, 'Exploiting partial information in taxonomy construction', in *Proc. of the 8th International Semantic Web Conference (ISWC 2009)*, (2009).

[17] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen, 'Scalable distributed reasoning using MapReduce', in *International Semantic Web Conference*, pp. 634–649, (2009).