Concurrent Classification of OWL Ontologies – An Empirical Evaluation

Mina Aslani and Volker Haarslev

Concordia University, Montreal, Quebec, Canada

Abstract. This paper describes our progress in developing algorithms for concurrent classification of OWL ontologies. We refactored the architecture of our research prototype and its employed algorithms by integrating lock-free data structures and adopting various optimizations to reduce overhead. In comparison to our earlier work we increased the size of classified ontologies by one order of magnitude, i.e., the size of processed ontologies is now beyond a quarter million of OWL classes. The main focus of this paper is an empirical evaluation with huge ontologies that demonstrates an excellent speedup that almost directly corresponds to the number of used processors or cores.

1 Introduction

Parallel algorithms for description logic (DL) 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 work on parallelizing DL reasoning [9] reported promising results using multi-core and multi-processor hardware, where the parallel treatment of disjunctions and individual merging (due to number restrictions) is explored. In [11] an approach on distributed reasoning for $\mathcal{ALCHIQ}$ is presented that is based on resolution techniques but does not address optimizations for TBox classification.

Other work has studied concurrency in light-weight ontology languages. There is a distributed Map Reduce approach algorithm for $\mathcal{EL}$+, however no experiments had been reported on the proposed algorithms [10]. Other work focuses on distributed reasoning, and these approaches are different than ours as they manage large-scale data which is beyond the memory of a single machine [11, 14, 6, 8, 12]. There also exists work on parallel distributed RDF inferencing (e.g., [13]) 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. Another work presents an optimized consequence-based procedure for classification of ontologies but it only addresses the DL $\mathcal{EL}$ [7].

The work in this paper is an extension of our work on Parallel TBox classification [1]. Compared to our previous work, this paper reports on an enhanced lock-free version of algorithms utilizing concurrency in a multi-core environment, optimizations that increase the performance, a performance evaluation with huge real-world ontologies in the range of 300K OWL classes (DL concepts) such as SNOMED. Our prototype not only addresses huge real-world ontologies but also does not compromise on DL complexity. It can process much more complex DLs (e.g., at least $\mathcal{SHIQ}$) than $\mathcal{EL}$, and provides an excellent speedup considering that no particular DL related optimization technique is used. The implemented prototype system performs concurrent TBox classification based on various parameters such as number of threads, size of partitions assigned to threads, and number of processors. Our evaluation demonstrates impressive performance improvements where the number of available processors almost linearly decreases the processing time due to a small overhead. It is important to note that the focus of this research is on exploring algorithms for concurrent TBox classification and not on developing a highly optimized DL reasoner. We are currently only interested in the speedup factor obtained from comparing sequential and parallel runs of our prototype.

2 The Concurrent TBox Classifier

This section describes the architecture of the implemented system and its underlying *sound and complete* algorithm for concurrent classification of DL ontologies. To compute the hierarchy in parallel, we developed a Java application using 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 number of processors. As thoroughly explained in [1], the program reads an input file containing a list of concept names to be classified and information about them which is generated by the OWL reasoner Racer [4]. Racer is only used for generating the input files for our prototype. The per-concept information available in the file includes the concept name, its parents (in the complete taxonomy), so-called told subsumers and disjoints, and pseudo model [5] information. This architecture was deliberately designed to facilitate our experiments by using existing OWL reasoners to generate auxiliary information and to make the Concurrent TBox Classifier independent of particular DLs.

The preprocessing algorithm uses a topological sorting similar to [1] and the order for processing concepts is based on the topologically sorted list. To manage concurrency and multi-threading in our system, as described in [1], a single-shared global tree approach is used. Also, to classify the TBox, two symmetric tasks are employed, i.e., the so-called enhanced tree traversal method [2] using top (bottom) search to compute the parents (children) of a concept to be inserted into the taxonomy.

In [1], we first introduced our algorithms for parallel classification and reported considerable performance improvements but we could only process relatively small ontologies. In this paper, we introduce the enhanced concurrent version of these algorithms, i.e., Algorithms 2, 6 and 7. In order to make the paper self-contained we repeat Algorithms 1, 3, 4 and 5 from [1].

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 with respect to the initial taxonomy created from 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 *located_concepts* indexed by thread

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
```

identifications. Using the Concurrency package in Java, synchronization on the nodes of the global tree as well as the entries in the global array have now been eliminated.

The procedure insert_partition inserts all concepts of a given partition into the global taxonomy. We use Concurrent collections from the java.util.concurrent package. This package supplies Collection implementations which are thread-safe and designed for use in multi-threaded contexts. Therefore, for updating a concept or its parents or children, no locking mechanism for the affected nodes of the global tree is needed anymore. Algorithm 2 first performs for each concept *new* the top-search phase (starting from the top concept ($\top$)) and possibly repeats the top-search phase for *new* if other threads updated the list of children of its parents. Then, it sets the parents of *new*. Afterwards the bottom-search phase (starting from the bottom concept ($\bot$)) is performed. Analogously to the top-search phase, the bottom search is possibly repeated and sets the children of *new*. After finishing the top and bottom search for *new*, the node *new* is added to the entries in *located_concepts* of all other busy threads; it is also checked whether other threads updated the entry in *located_concepts* for this thread. If this was the case, the top and/or bottom search need to be repeated correspondingly.

To reduce overhead in re-running of top or bottom search, we only re-run twice. If the concept *new* is still not ready to be inserted; e.g., there is any interaction between *new* and a concept in *located_concepts*; it will be added to the partition list of concepts (to be located later), and also eliminated from the other busy threads' *located_concepts* list, otherwise, *new* can be inserted into the taxonomy using Algorithm 7. 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. Algorithms 3 and 4[1] outline the traversal procedures for the top-search phase.

The possible incompleteness caused by parallel classification [1] can be characterized by the following two scenarios: *Scenario I*: In top search, as the new concept is 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 concept *new* is not aware of the recent change and this might cause missing subsumptions if there is any interaction between the concept *new* and the added children. The same might happen in bottom search if the bottom search for the concept *new* is not informed of the recent change to the list of parents of the current node. *Scenario II*: Between the time that top search has been started to find the location of the concept *new* in the taxonomy and the time that its location has been decided, another thread has

[1] Algorithm found_in_ancestors(*current*,*new*) checks if *current* is an ancestor of *new*.

Algorithm 2 insert_partition(*partition,id*)

```
for all $new \in partition$ do
  $rerun \leftarrow 0$
  $finish\_rerun \leftarrow$ false
  $parents \leftarrow$ top_search($new$,$\top$)
  while $\neg$ consistent_in_top_search($parents$,$new$) do
    $parents \leftarrow$ top_search($new$,$\top$)
  predecessors($new$) $\leftarrow parents$
  $children \leftarrow$ bottom_search($new$,$\bot$)
  while $\neg$ consistent_in_bottom_search($children$,$new$) do
    $children \leftarrow$ bottom_search($new$,$\bot$)
  successors($new$) $\leftarrow children$
  for all busy threads $t_i \neq id$ do
    $located\_concepts(t_i) \leftarrow located\_concepts(t_i) \cup \{new\}$
  $check \leftarrow check\_if\_concept\_has\_interaction(new, located\_concepts(id))$
  while ($check \neq 0$) and $\neg finish\_rerun$ do
    if $rerun < 3$ then
      if $check = 1$ then
        $new\_predecessors \leftarrow$ top_search($new$,$\top$)
        $rerun \leftarrow rerun + 1$
        predecessors($new$) $\leftarrow new\_predecessors$
      if $check = 2$ then
        $new\_successors \leftarrow$ bottom_search($new$,$\bot$)
        $rerun \leftarrow rerun + 1$
        successors($new$) $\leftarrow new\_successors$
      $check \leftarrow check\_if\_concept\_has\_interaction(new, located\_concepts(id))$
    else
      $finish\_rerun \leftarrow$ true
      for all busy threads $t_i \neq id$ do
        $located\_concepts(t_i) \leftarrow located\_concepts(t_i) \setminus \{new\}$
  if $\neg finish\_rerun$ then
    insert_concept_in_tbox($new$, predecessors($new$), successors($new$))
```

placed at least one concept into the hierarchy which the concept *new* has an interaction with. Again, this might cause missing subsumptions and is analogously also applicable to bottom search.

Both scenarios are properly addressed in Algorithm 2 to ensure completeness. Every time a thread locates a concept in the taxonomy, it notifies the other threads by adding this concept name to their "located_concepts" list. Therefore, as soon as a thread finds the parents and children of the concept *new* by running top_search and bottom_search; it checks if there is any interaction between concept *new* and the concepts located in the "located_concepts" list. Based on the interaction, top_search or bottom_search needs to be repeated accordingly. If no possible situations for incompleteness are discovered anymore, Algorithm 7 is called. To resolve the possible incompleteness we utilize Algo-

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

```
mark(current,‘visited’)
pos-succ ← $\emptyset$
captured_successors(new)(current) ← successors(current)
for all $y \in$ successors(current) do
    if enhanced_top_subs(y,new) then
        pos-succ ← pos-succ $\cup \{y\}$
if pos-succ $= \emptyset$ then
    return {current}
else
    result ← $\emptyset$
    for all $y \in$ pos-succ do
        if y not marked as ‘visited’ then
            result ← result $\cup$ 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 $z \in$ predecessors(current)
            enhanced_top_subs(z,new)
        and found_in_ancestors(current,new) then
    mark(current,‘positive’)
    return true
else
    mark(current,‘negative’)
    return false
```

rithms 5 and 6.[2] The procedure consistent_in_bottom_search is not shown here because it mirrors consistent_in_top_search.

3 Evaluation

In the previous section, we explained the algorithms used in our Concurrent TBox Classifier. In this section, we study the scalability and performance of our prototype. Here, we would like to explain the behavior of our system when we run it in a (i) sequential or (ii) parallel multi-processor environment. We also describe how the prototype performs when we have huge real-world ontologies with different DL complexities. Therefore, in the remaining of this section, we report on the conducted experiments.

We first provide a description of the used platform and the implemented prototype, then we describe the test cases used to evaluate Concurrent TBox Classifier and provide

[2] Algorithm interaction_possible(*new*,*concept*) uses pseudo model merging [5] to decide whether a subsumption is possible between *new* and *concept*.

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

```
for all $pred \in parents$ do
    if successors($pred$) $\neq$ captured_successors($new$)($pred$) then
        $diff \leftarrow$ successors($pred$) $\setminus$ captured_successors($new$)($pred$)
        for all $child \in diff$ do
            if found_in_ancestors($child$,$new$) then
                return false
return true
```

Algorithm 6 check_if_concept_has_interaction(*new,located_concepts*)

```
The return value indicates whether and what type of re-run needs to be done:
0 : No re-run in needed
1 : Re-run TopSearch because a possible parent could have been overlooked
2 : Re-run BottomSearch because a possible child could have been overlooked
if located_concepts = $\emptyset$ then
    return 0
else
    for all concept $\in$ located_concepts do
        if interaction_possible(new,concept) then
            if found_in_ancestors(new,concept) then
                return 2
            else
                return 1
        else if interaction_possible(concept,new) then
            if found_in_ancestors(new,concept) then
                return 2
            else
                return 1
    return 0
```

an overview of the parameters used in the experiments. Finally, we show the results and discuss the performance of the classifier. In addition, the measured runtimes in the figures are shown in seconds using a logarithmic scale.

Platform and implementation All the experiments were conducted on a high performance parallel computing cluster. The nodes in the cluster run an HP-version of RedHat Enterprise Linux for 64 bit processors, with HP's own XC cluster software stack. To evaluate our approach, Concurrent TBox Classifier has been implemented in Java using lock-free data structures from the java.util.concurrent package with minimal synchronization.

Test cases Table 1 shows a collection of 9 mostly publicly available real-world ontologies. Note that the chosen test cases exhibit different sizes, structure, and DL complexities. The benchmark ontologies are characterized by their name, size in number of named concepts or classes, and used DL.

Parameters used in experiments The parameters used in our empirical evaluation and their meaning are described below (the default parameter value in shown in bold).

Algorithm 7 insert_concept_in_tbox(*new,predecessors,successors*)

```
for all $pred \in predecessors$ do
    successors($pred$) $\leftarrow$ successors($pred$) $\cup \{new\}$
for all $succ \in successors$ do
    predecessors($succ$) $\leftarrow$ predecessors($succ$) $\cup \{new\}$
```

Table 1. Characteristics of the used test ontologies (e.g., $\mathcal{LH}$ denotes the DL allowing only conjunction and role hierarchies, and unfoldable TBoxes)

Ontology	DL language	No. of named concepts
Embassi-2	$\mathcal{ALCHN}$	657
Galen1	$\mathcal{ALCH}$	2,730
LargeTestOntology	$\mathcal{ELHR}+$	5,584
Tambis-2a	$\mathcal{ELH}$	10,116
Cyc	$\mathcal{LHF}$	25,566
EClass-51En-1	$\mathcal{LH}$	76,977
Snomed-2	$\mathcal{ELH}$	182,869
Snomed-1	$\mathcal{ELH}$	223,260
Snomed	$\mathcal{ELH}$	379,691

– *Number of Threads*: To measure the scalability of our system, we have performed our experiments using different numbers of threads (**1**, 2, 4).
– *Partition Size*: The number of concepts (**5**, 25) that are assigned to every thread and are expected to be inserted by the corresponding thread. Similar to the number of threads, this parameter is also used to measure the scalability of our approach.
– *Number of Processors*: For the presented benchmarks we always had 8 processors or cores[3] available.

Performance In order to test the effect of these parameters in our system, the benchmarks are run with different parameter values. The performance improvement is measured using the speedup factor which is defined as $Speedup_p = \frac{T_1}{T_p}$, where $Speedup_p$ is the speedup factor, and

– p is the number of threads. In the cluster environment we always had 8 cores available and never used more than 8 threads in our experiments, so, each thread can be considered as mapped to one core exclusively;
– T_1 is the CPU time for the sequential run using only one thread and one single partition containing all concept names to be inserted;
– T_p is the CPU time for the parallel run with p threads.

Effect of changing only the number of threads To measure the performance of the classifier in this case we selected EClass-51En-1 as our test case and ran the tests with a fixed partition size (5 or 25) but a different number of threads (2 and 4), as shown in Fig. 1. In the following we use $P_{threads,partition\_size}$ to indicate a parallel multi-core setting where the subscripts give the number of cores available, the number of threads

[3] For ease of presentation we use the terms *core* and *processor* as synonyms here.

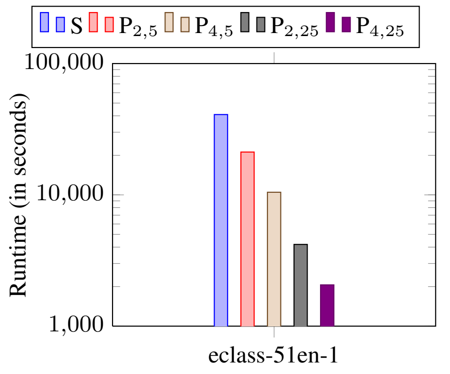

Fig. 1. Runtimes for eclass-51en-1 using 5 settings: S (sequential), $P_{2,5}$, $P_{4,5}$, $P_{2,25}$, $P_{4,25}$ ($P_{threads,partition\_size}$)

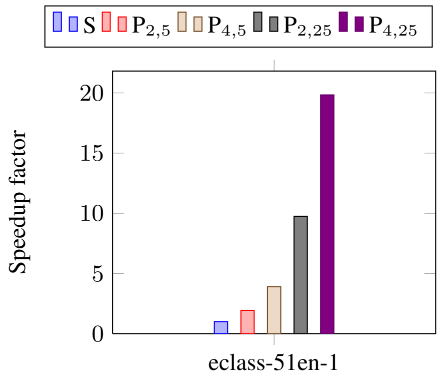

Fig. 2. Speedup for eclass-51en-1 from Fig. 1

created, and the partitions size used (from left to right). In the test cases $P_{2,5}$ and $P_{4,5}$, we get an ideal speedup proportional to the number of threads, as shown in Fig. 2. As we can see, doubling the number of threads from S to $P_{2,5}$ and to $P_{4,5}$, each time doubles the speedup, in other words, decreases the CPU time by the number of threads. This is the ideal speedup that we were expecting to happen.

Comparing the test cases S, $P_{2,25}$, and $P_{4,25}$, we get an even better speedup, also shown in Fig. 1 and 2. In this case, the CPU time decreases almost to $\frac{1}{10}$ compared to the sequential case (S). This speedup is due to a combination of the partition size as well as the cache effect and results from the different memory hierarchies of a cluster with modern computers. When we increase the number of threads to 4, the speedup is again proportional to the number of threads and this is what we expected. Here, by doubling the number of threads, the speedup doubles.

Effect of changing only partition sizes The performance of the classifier in this case for EClass-51En-1 is also shown in Fig. 1 with a fixed number of threads (2 or 4) but different partition sizes (5 or 25). When using 2 threads, compared to case S, we get the ideal speedup for $P_{2,5}$, as shown in Fig. 2. As we can see, doubling the number of threads, doubles the speedup, in other words, decreases the CPU time by half. This is the ideal case which is what we were expecting to happen. Again, compared to case S if the partition size is increased to 25, it shows the same speedup as shown in Fig. 2. In this case, the CPU time decreases almost to $\frac{1}{5}$ compared to the previous case.

In the scenario with 4 threads, we get a corresponding speedup, as shown in Fig. 2. In this case, the CPU time decreases to almost $\frac{1}{10}$ compared to the sequential case. This speedup again is due to a combination of the number of threads as well as the cache effect. When we increase the partition size to 25, the speedup is what we expected. Here, by multiplying the partition size by 5, the speedup is multiplied by five too.

Increasing the partition size, means that more concepts are assigned to one thread; therefore, all the related concepts are inserted into the taxonomy by one thread. Hence, increasing the partition size, reduces the number of corrections.

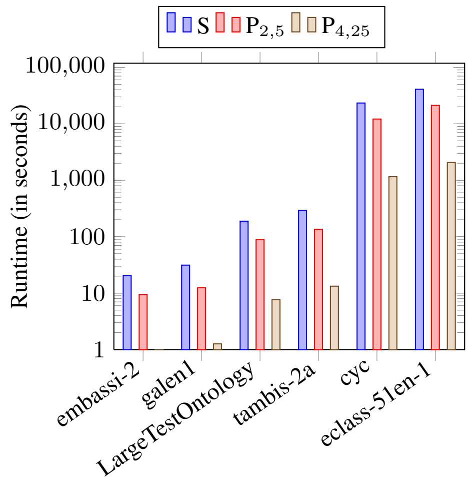

Fig. 3. Runtimes for ontologies using 3 settings: S (sequential), $P_{2,5}$, $P_{4,25}$

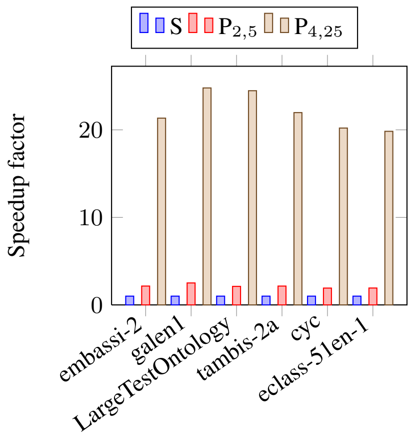

Fig. 4. Speedup for ontologies from Fig. 3

Effect of increasing both the number of threads and the partition size In this scenario, we measured the CPU time when increasing both the number of threads and the partition size. In Fig. 3 and 4, our test suite includes the ontologies Embassi-2, Galen1, LargeTestOntology, Tambis-2a, Cyc, and EClass-51En-1. The CPU time for each test case is shown in Fig. 3 and the speedup factor for each experiment is depicted in Fig. 4. As the results show, in the scenario with 2 threads and partition size 5, the speedup doubles compared to the sequential case and is around 2 and this is what we were expecting. When we increase the number of threads as well as the partition size, for the scenario with 4 threads and partition size 25, the CPU time decreases dramatically and therefore the speedup factor is above 20 for most test cases. This is more than a linear speedup, and it is the result of increasing the thread number as well as partition size together with the cache effect. The highest speedup factor is reported with test case Galen1.

Experiment on very large ontologies We selected 3 Snomed variants as very large ontologies with more than 150,000 concepts. Snomed-2 with 182,869 concepts, Snomed-1 with 223,260 concepts, and Snomed with 379,691 concepts were included in our tests. Fig. 7 shows an excellent improvement of CPU time for the parallel over the sequential case. In Fig. 8, the speedup factor is almost 2, which the expected behavior. The best speedup factor is observed for test case Snomed.

Observation on the increase of size of ontologies We chose Cyc, EClass-51en-1, Snomed-1, Snomed-2, and Snomed as test cases. Here, as shown in Fig. 5 and 7, in a parallel setting with 2 threads, the CPU time is divided by 2 compared to the sequential case. The speedup, shown in Fig. 6 and 8, is linear and is consistent for our benchmark ontologies even when the size of the ontologies increases.

Overall, the overhead is mostly determined by the quality of the told subsumers and disjoints information, the imposed order of traversal within a partitioning, the division of the ordered concept list into partitions, and the number of corrections which have been taken place (varied between 0.5% and 3% of the ontology size; depends on the

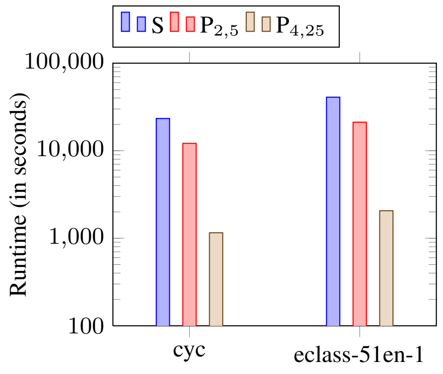

Fig. 5. Runtimes for cyc and eclass-51en-1 using 3 settings: S (sequential), $P_{2,5}$, $P_{4,25}$

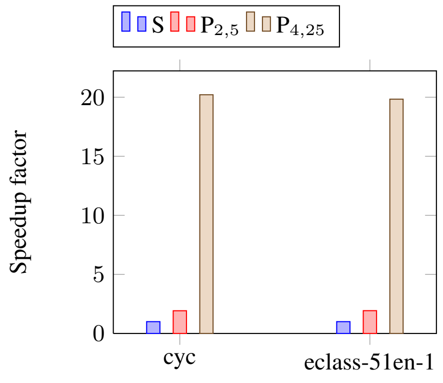

Fig. 6. Speedup for ontologies from Fig. 5

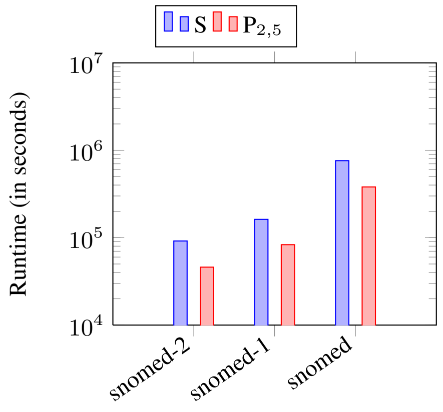

Fig. 7. Runtimes for snomed using 2 settings: S (sequential), $P_{2,5}$

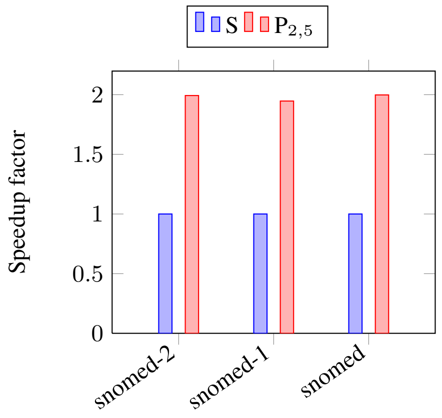

Fig. 8. Speedup for ontologies from Fig. 7

structure of ontology as well as the number of threads and partition size). In general, one should try to insert nodes as close as possible to their final order in the tree using a top to bottom strategy.

In Concurrent TBox Classifier no optimization techniques for classification have been implemented. For instance, there are well-known optimizations which can avoid subsumption tests or eliminate the bottom search for some DL languages or decrease the number of bottom searches in general. Of course, our system is not competitive at all compared to highly optimized DL reasoners or special-purpose reasoners designed to take advantage of the characteristics of the $\mathcal{EL}$ fragment (e.g., see [7]). In our case, we can easily classify ontologies that are outside of the $\mathcal{EL}$ fragment.

4 Conclusion

In this paper, we have shown an excellent scalable technique for concurrent OWL ontology classification. The explained architecture, which proposes lock-free algorithms with limited synchronization, utilizes concurrency in a multi-core environment. The experimental results show the effectiveness of our algorithms. We can say that this work appears to be the first which documents significant performance improvements in a multi-core environment using real-world benchmarks for ontologies of various DL complexities.

References

1. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics - first experimental results. In: Proceedings of the 19th European Conference on Artificial Intelligence - ECAI 2010, Lisbon, Portugal, Aug. 16-20, 2010. pp. 485–490 (2010)
2. Baader, F., Franconi, E., Hollunder, B., Nebel, B., Profitlich, H.: An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. Applied Artificial Intelligence 4(2), 109–132 (1994)
3. Bergmann, F., Quantz, J.: Parallelizing description logics. In: Proc. of 19th Ann. German Conf. on Artificial Intelligence. pp. 137–148. LNCS, Springer-Verlag (1995)
4. Haarslev, V., Möller, R.: RACER system description. In: Proc. of the Int. Joint Conf. on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy. pp. 701–705. LNCS (Jun 2001)
5. Haarslev, V., Möller, R., Turhan, A.Y.: 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, Siena, Italy. pp. 61–75 (2001)
6. Hogan, A., Pan, J., Polleres, A., Decker, S.: SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples. In: Proc. 9th Int. Semantic Web Conf. pp. 337–353 (2010)
7. Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent classification of EL ontologies. In: Proc. of the 10th Int. Semantic Web Conf. pp. 305–320 (2011)
8. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: Proc. 19th Int. Conf. on World Wide Web. pp. 531–540 (2010)
9. Liebig, T., Müller, F.: 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. LNCS, vol. 4806, pp. 1135–1144. Springer-Verlag (2007)
10. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for EL+. In: Proc. 23rd Int. Workshop on Description Logics. pp. 464–474 (2010)
11. Schlicht, A., Stuckenschmidt, H.: 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)
12. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: WebPIE: a webscale parallel inference engine using mapreduce. In: J. of Web Semantics (2011)
13. Urbani, J., Kotoulas, S., Oren, E., van Harmelen, F.: Scalable distributed reasoning using MapReduce. In: International Semantic Web Conference. pp. 634–649 (2009)
14. Weaver, J., Hendler, J.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Proc. 8th Int. Semantic Web Conf. pp. 87–101 (2009)