A Parallel Reasoner for the Description Logic $\mathcal{ALC}$

Kejia Wu and Volker Haarslev

Concordia University, Montreal, Canada

Abstract. Multi-processor/core systems have become ubiquitous but the vast majority of OWL reasoners can process ontologies only sequentially. This observation motivates our work on the design and evaluation of $\mathfrak{Deslog}$, a parallel tableau-based description logic reasoner for $\mathcal{ALC}$. A first empirical evaluation for TBox classification demonstrates that $\mathfrak{Deslog}$'s architecture supports a speedup factor that is linear to the number of utilized processors/cores.

1 Introduction

The popularity of multi-processor/core computing facilities makes *Description Logic (DL)* reasoning feasible that scales w.r.t. the number of available cores.[1] Modern many-core architectures together with operating systems implementing symmetric multitasking efficiently support thread-level parallelism (TLP), where smaller subtasks are implemented as threads that are concurrently executed on available cores. In order to utilize such hardware for making DL reasoning scalable to the number of available cores, one has to develop new reasoning architectures efficiently supporting concurrency. Moreover, many well-known tableau optimization techniques need to be revised or adapted in order to be applicable in a parallel context. In this paper we present a new DL reasoning framework that fully utilizes TLP on many-core systems. In principle, standard tableau algorithms are well suited for parallelization because tableau completion rules usually do not depend on a sequential execution but require shared access to common data structures.

We consider the inherent non-determinism of DL tableaux as a feature of DL reasoning that naturally leads to parallel algorithms which are suitable for a shared-memory setting. For instance, a source for such a non-determinism are disjunctions and qualified cardinality restrictions for logics containing at least $\mathcal{ALCQ}$. Many standard DL reasoning services, e.g. concept satisfiability testing, TBox classification, instance checking, ABox realization, etc. might be amenable to be implemented in a non-deterministic way supporting parallelism. However, the use of parallelism in DL reasoning has yet to be explored systematically. Although it is advantageous to seek scalable solutions by means of parallelism, little progress has been made in parallel DL reasoning during the last decade [6]. Most DL reasoning algorithms and optimization techniques have only been investigated in a *sequential* context, but how these methods should be accommodated to parallelism remains mostly open. Besides these algorithmic and theoretical issues, the efficient implementation of a scalable parallel DL reasoner is also worth to be investigated. Many practical issues on shared-memory parallelism need to

[1] For ease of presentation we consider the terms processor and core as synonyms.

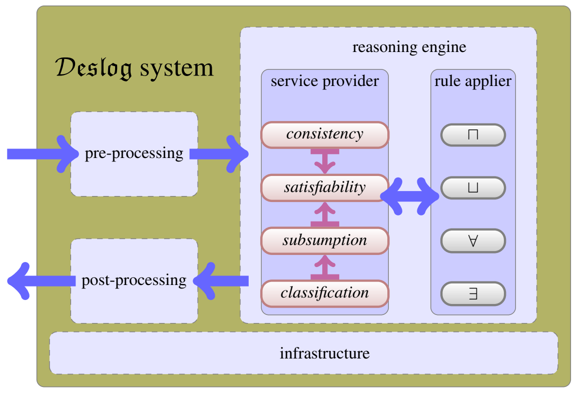


Fig. 1: The framework of Deslog

be researched, e.g., selecting suitable parallel architectures, designating efficient data structures and memory management, and exploring parallel algorithms.

In the remaining sections we introduce the design of Deslog, discuss related work, and present an evaluation demonstrating a mostly (super)linear scalability of Deslog.

2 Architecture of Deslog

2.1 Framework

The shared-memory parallel reasoner Deslog consists of three layers: (i) *pre-processing* layer, which converts the *Web Ontology Language (OWL)* representation of ontologies to internal data structures; (ii) *reasoning engine* layer, which performs the standard DL reasoning services and is composed of two key components, the *service provider* and the *tableau rule applier*; (iii) *post-processing* layer, which collects, caches, and saves reasoning results; (iv) *infrastructure* layer, which provides core components and utilities, such as structures representing concepts and roles, and the object copy tool. Figure 1 gives an overview of the framework.

First, OWL ontology data is read into the pre-processing layer. Various typical pre-processing operations, such as transformation into *negation normal form (NNF)*, axiom re-writing, and axiom absorption, are executed in this layer. The reasoner's run-time options, such as service selection, maximum number of threads, and rule application order, are also set up on this layer. We implemented this layer by using the OWL API [14]. The pre-processed data is streamed to the reasoning engine.

The reasoning engine performs primarily inference computation. The first key component of the reasoning engine is the service provider. As with popular DL reasoning systems, Deslog provides standard reasoning services, such as testing TBox consistency, concept satisfiability, etc. As we know, these services may depend on each other. In Deslog, the classification service depends on subsumption, and the latter depends on the satisfiability service. The service provider uses a suite of tableau-based algorithms to perform reasoning. The reasoner adopts tableaux as its primary reasoning method, which is complemented by another key component, the tableau expansion rule applier. The main function of this component is to execute tableau rules in some order to build expansion forests. In $\mathcal{D}\mathfrak{eslog}$, tableau expansion rules are designed as configurable plug-ins, so what rule has to be applied in what application order can be specified flexibly. At present, $\mathcal{D}\mathfrak{eslog}$ implements the standard $\mathcal{ALC}$ tableau expansion rules [4].

All three layers mentioned above use the facilities provided by the infrastructure layer. All common purpose utilities are part of this layer. For example, the threads manager, global counters, and the *globally unique identifier (GUID)* generator. In addition, the key data structures representing DL elements and basic operations on them are provided by this layer.

2.2 Key Data Structures

In contrast to popular DL reasoning systems, $\mathcal{D}\mathfrak{eslog}$ aims to improve reasoning performance by employing parallel computing, while data structures employed by sequential DL reasoners are not always suitable for parallelism.

Tree structures have been adopted by many tableau-based reasoners. However, a naive tree data structure introduces data races in a shared-memory parallel environment and are not well suited in a concurrency setting. Therefore, we need to devise more efficient data structures in order to reduce the amount of shared data as much as possible.

The new data structures facilitating concurrency must support DL tableaux. One important function of trees is to preserve non-deterministic branches generated during tableau expansion. Non-deterministic branches are mainly produced by the *disjunction rule* and the *at-most number restriction rules*. To separate non-deterministic branches into independent data vessels, which are suited to be processed in parallel, we adopt a list-based structure, called *stage*, to maintain a single non-deterministic branch, and a queue-based structure, called *stage pool*, to buffer all branches in a tableau. Every stage is composed of the essential elements of a DL ontology, concepts and roles.

As with any DL reasoner, the representation of concepts and roles are fundamental design considerations. The core data structure of $\mathcal{D}\mathfrak{eslog}$ is a four-slot list representing a *concept*. A *literal* uniquely identifies a distinct concept. An *operator* indicates the dominant DL constructor applied to a concept. Available constructors cover intersection, disjunction, existential and value restriction, and so on, and this slot can also be empty. The remaining two slots hold pointers to extend nested concept definitions, namely *left* and *right*. Figure 2 illustrates a DL concept encoded with the $\mathcal{D}\mathfrak{eslog}$ protocol.

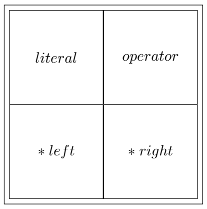


Fig. 2: 𝔇𝔢𝔰𝔩𝔬𝔤 data structure for a concept

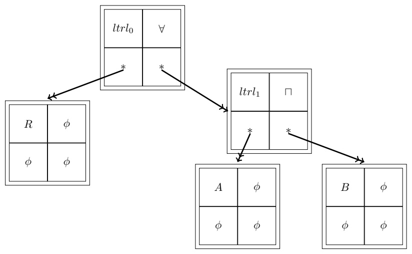
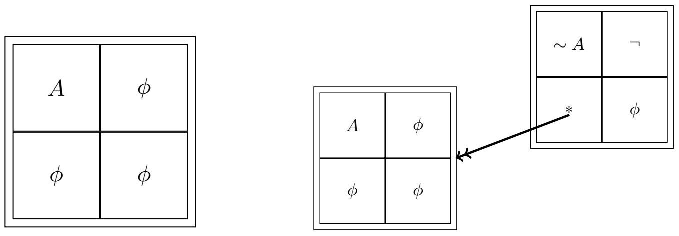


Fig. 3: 𝔇𝔢𝔰𝔩𝔬𝔤 data structure for the DL expression $\forall R.(A \sqcap B)$

Roles in 𝔇𝔢𝔰𝔩𝔬𝔤 are handled as a special type of concepts and have a similar structure as concepts. For instance, the encoding of the DL expression $\forall R.(A \sqcap B)$ is shown in Figure 3. Further properties needed for describing a role can be added to the generic structure, e.g., the number restriction quantity. A role data structure is also associated with a list recording instance pairs. With this design, DL concepts can be lined up seamlessly. Instances (i.e., labels in tableau expansions) are lists holding their typing data, concepts. There are also helper facilities, such as a role pool and an instance pool, which are useful to accelerate the indexing of objects.

A notable point on our encoding is how the *complement* of an atomic concept (i.e., concept name) is expressed (see Figures 4a and 4b).



(a) Atomic concept A (b) Atomic concept $\neg A$

Fig. 4: 𝔇𝔢𝔰𝔩𝔬𝔤 data structure—atomic concepts

A principle of 𝔇𝔢𝔰𝔩𝔬𝔤's design is to model objects and behaviours involved in DL reasoning as independent abstractions as much as possible in order to facilitate concurrent processing. For instance, branches created during tableau expansion are encapsulated into standalone objects. Thus, a whole tableau expansion forest is designed as a list of branch objects. Tableau expansion rules and even some key optimization techniques are also designed as independent components.

2.3 Implementation

As aforementioned, multi-processor computers are becoming the main stream, it is expected that idle processors are utilized, and shared-memory TLP is quite suitable for this purpose.

One significant aspect of this research is to investigate how well important DL reasoning optimization techniques are suited to be implemented in a parallel reasoner and how they should be adapted if plausible. $\mathcal{D}\mathfrak{eslog}$ has adopted the following optimization techniques.

1. **Lazy-unfolding** This technique enables a reasoner to unfold a concept only when necessary [5].
2. **Axiom absorption** *Disjunctive branches* introduced by naively internalizing TBox-axioms is one of the primary sources of reasoning inefficiency. With the axiom absorption technique, a TBox is separated into two parts, the *general* TBox and the *unfoldable* TBox. Then, using internalization to process the general part and lazy-unfolding to process the unfoldable part can reduce reasoning time dramatically [15, 25].
3. **Semantic branching** This DPLL style technique prunes disjunctive branches by avoiding to compute the same problem repeatedly [15].

Other primary optimization techniques, such as *dependency directed backtracking* [4, Chapter 9] [15] and *model merging* [12] are currently being implemented. It is noticeable that not all significant optimization techniques are suitable for concurrency. Some of them still depend on complex shared data and may significantly degrade the performance of a concurrent program. Based on these elemental techniques, we completed a suite of standard TBox reasoning services.

The current system implements a parallel $\mathcal{ALC}$ TBox classifier. It can concurrently classify an $\mathcal{ALC}$ terminology. The parallelized classification service of $\mathcal{D}\mathfrak{eslog}$ computes subsumptions in a brutal way [5]. It is obvious that the algorithm is sound and complete and has order of n^2 time complexity in a sequential context. In order to figure out a terminology hierarchy, the algorithm calculates the subsumptions of all atomic concepts pairs. A subsumption relationship only depends on the involved concepts pair, and does not have any connections with the computation order. Therefore, the subsumptions can be computed in parallel, and soundness and completeness are retained in a concurrent context.

A rather difficult issue in implementing a parallel DL reasoner is managing overhead. This issue is relatively easy for high level parallel reasoning, where multiple threads mainly execute reading operations on some shared data, thus, we implemented the parallel classification service first.

Besides the high-level parallelized service, classification, low level parallelized processing is being developed. In the architecture of $\mathcal{D}\mathfrak{eslog}$, the classification service uses subsumption, and subsumption uses satisfiability. The low level parallel reasoning focuses on dealing with non-deterministic branches, which are represented as stages in $\mathcal{D}\mathfrak{eslog}$.

It might seems easy to process stages in parallel, but quite some effort is required to achieve a satisfying scalability via concurrency. The first noticeable fact is that from a root stage every stage may generate new ones. At present, our strategy is using one thread to process one stage. That means the stages buffer, the stage pool, is frequently accessed by multiple threads. That accessing includes both writing and reading shared data frequently. So, designing a high-performance stage buffer and efficient accessing schemes is an essential condition for the enabling a of scalable performance improvement. Otherwise, a parallel approach only causes overhead instead of performance improvement. We are currently working on efficient low-level parallel reasoning.

Although there are robust shared-memory concurrent libraries available, such as the C++ *Boost.Thread* library and the Java *concurrent* package, according to our experience, using these libraries immoderately often degrades performance. Therefore, one needs to design sophisticated structures which better avoid shared data, or which do not access shared data frequently.

3 Evaluation

$\mathcal{D}\mathfrak{eslog}$ is implemented in Java 6 in conformity with the aforementioned design. The *parallelism* of $\mathcal{D}\mathfrak{eslog}$ is based on a *multi-threading model* and aims at exploiting *symmetric multiprocessing (SMP)* supported by *multi-processor computing* facilities. The system is implemented in Java 6 for Java's relatively mature parallel ecosystem.[2] Specifically, the *java.util.concurrent* package of Java 6 is utilized.

In the following we report on experiments to demonstrate that a shared-memory parallel tableau-based reasoner can achieve a scalable performance improvement.

3.1 Conducted Experiments

The classification service of $\mathcal{D}\mathfrak{eslog}$ can be executed concurrently by multiple threads. We conducted a group of tests, and they show that $\mathcal{D}\mathfrak{eslog}$ has an obvious scalability.

All tests were conducted on a 16-core computer running Solaris OS and Sun Java 6. Many of the test cases were chosen from *OWL Reasoner Evaluation Workshop 2012 (ORE 2012)* data sets. We manually changed the expressivity of some test cases to $\mathcal{ALC}$ so that $\mathcal{D}\mathfrak{eslog}$ could reason about them. Table 1 lists the metrics of the test cases. The results are shown in Figures 5-7.

3.2 Discussion

The experimental results collected from testing ontologies show a scalable performance improvement. The tests on more difficult ontologies demonstrate a better scalability due to reduced overhead.

Because the computing time of the single thread configuration, T_1, is rather small for some ontologies, often smaller than ten seconds, the overhead introduced by maintaining multiple threads can limit the scalability. For some of these ontology tests, the reasoning times are reduced to several milliseconds, i.e., the whole work load assigned to a single thread is around several milliseconds in these settings. According to our empirical results, a small work load w.r.t. the overhead, which is produced by manipulating

[2] All components and sources of the system are available at *http://code.google.com/p/deslog/*.

Table 1: Characteristics of the used test onologies

Ontology	**DL expressivity**	**Concept count**	**Axiom count**
bfo	$\mathcal{ALC}$	36	45
pharmacogenomics_complex	$\mathcal{ALC}$	145	259
economy	$\mathcal{ALCH(D)}$	339	563
transportation	$\mathcal{ALCH(D)}$	445	489
mao	$\mathcal{ALE}+$	167	167
yeast_phenotype	$\mathcal{AL}$	281	276
loggerhead_nesting	$\mathcal{ALE}$	311	347
spider_anatomy	$\mathcal{ALE}$	454	607
pathway	$\mathcal{ALE}$	646	767
amphibian_anatomy	$\mathcal{ALE}+$	703	696
flybase_vocab	$\mathcal{ALE}+$	718	726
tick_anatomy	$\mathcal{ALE}+$	631	947
plant_trait	$\mathcal{ALE}$	976	1140
evoc	$\mathcal{AL}$	1001	990
protein	$\mathcal{ALE}+$	1055	1053

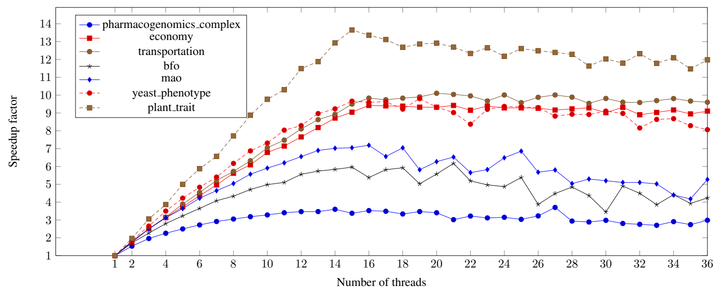


Fig. 5: Speedup factor for pharmacogenomics_complex, economy, transportation, bfo, mao, yeast_phenotype, and plant_trait

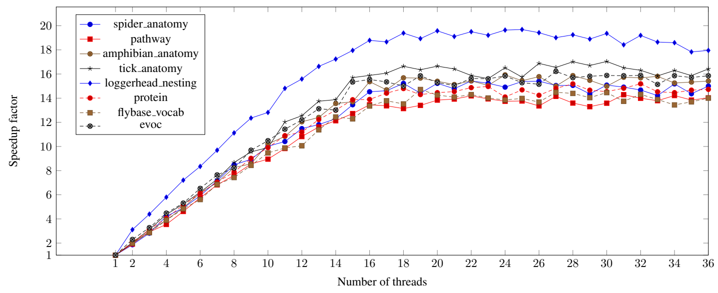


Fig. 6: Speedup factor for spider_anatomy, pathway, amphibian_anatomy, tick_anatomy, loggerhead_nesting, protein, flybase_vocab, and evoc

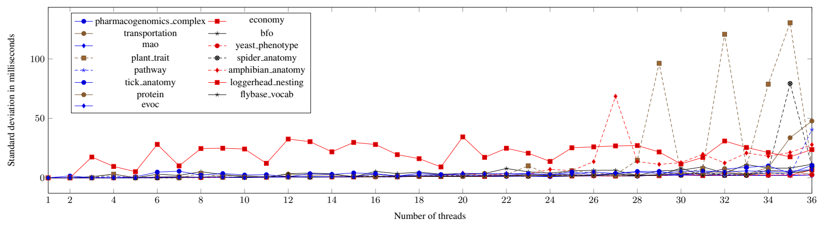


Fig. 7: The standard deviations of thread runtimes

threads as well as accessing shared data, counteracts the benefits gained from parallel processing and the reasoning performance starts to degrade.

When testing ontologies that are big enough, the observed scalability is linear, sometimes even superlinear. These bigger ontologies need longer single thread computing times (T_1). The overhead introduced by maintaining a *tolerable* number of multiple threads is very small and becomes insignificant. A tolerable number, N_i, should always be smaller than or equal to the total number of available processors. In our conducted tests due to the available hardware we have $N_i \in [1, 16]$ but in order to stress multi-threading we conducted all tests with up to 32 threads. In some cases, even though the number of threads exceeds 16, the reasoning performance keeps stable in a rather long run. This clearly supports our hypothesis that a further scalability improvement could be achieved by adding more processors, and we will verify this hypothesis when better experimental hardware becomes available.

Figure 7 shows the standard deviation of thread runtimes measured in the series of tests (in the unit of milliseconds). Overall, the deviations are limited to an acceptable range, i.e., below 140 milliseconds, which is relatively insignificant w.r.t. system overhead. This implies that the work load is well balanced among threads. That is to say, all threads are as much busy as possible. For the most part, when the number of threads is smaller than the tolerable number, 16, deviations are normally close to 0. When threads are added beyond 16, deviations become greater. This is because some processors execute more than one thread, and hereby the thread contexts switching produces a lot of overhead. In our original implementation, we had distributed all subsumption candidates into independent lists, every of which mapped to a thread, but the deviations were sometimes too large. So, the $\mathfrak{Deslog}$ classification uses now a shared queue to buffer all subsumption candidates, in order keep all threads busy.

We had conducted similar experiments on a high-performance computing cluster, and the results were rather disappointing. The speedup factor gained on the cluster was generally below 3 although we assigned at least 16 processors for each test. The most plausible explanation we can give is that the complex hardware and software environment of the cluster degrades the performance of $\mathfrak{Deslog}$. The cluster consists of three types of computing nodes with respect to the built-in processors: 4-core, 8-core, and 16-core. And the same type of computers may have heterogeneous architectures. A job is scheduled on one or more computers randomly. It is normal that a job is assigned to more than one computer, and the communication between computers results in a bottleneck. Another possible reason is that the cluster does not guarantee exclusive usage, which means it is possible that more than one job is running on the same computer at the same time.

$\mathfrak{Deslog}$ can only deal with $\mathcal{ALC}$ ontologies at present, and we expect that more interesting results will be obtained by implementing more powerful tableau expansion rules (e.g., see [16]). We already investigated the feasibility of several tableau optimization techniques for a concurrency setting, but most of them are not yet fully implemented or tested.

4 Related Work

Our research investigates the potential contribution of concurrent computing to tableau-based DL reasoning. Tableau-based DL reasoning has been extensively researched in sequential contexts. A large amount of literature is available for addressing sequential DL reasoning (see [4]). Only a few approaches investigated parallel DL reasoning so far.

The work in [19] reports on a parallel $\mathcal{SHN}$ reasoner. This reasoner implemented parallel processing of *disjunctions* and *at-most cardinality restrictions*, as well as some DL tableau optimization techniques. The experimental results show noticeable performance improvement in comparison with sequential reasoners. This work mainly showed the feasibility of parallelism for low level tableau-based reasoning, and did not mention high level reasoning tasks, such as classification. Besides utilizing non-determinism, this research also presented the potential of making use of *and-parallelism*, and we plan to follow up on this idea.

The canonical top-search algorithm, as well as its dual bottom-search, can be executed in parallel, but extra work is needed to preserve completeness. Such an approach is presented in [1, 2] and the experimental results are very promising and demonstrate the feasibility of parallelized DL reasoning.

In [17, 23] a *consequence-based* DL reasoning method was proposed, mainly dealing with Horn ontologies. Based on consequence-based reasoning and the results of [3], the work in [18] reports on a highly optimized reasoner that can classify $\mathcal{EL}$ ontologies *concurrently*.

Two hypothesises on parallelized ontology reasoning were proposed in [6]: *independent ontology modules* and *a parallel reasoning algorithm*. *Independent ontology modules* strive for structuring ontologies as modules which naturally can be computed in parallel. The idea of partitioning ontologies into modules is supported by [11, 9, 10]. According to the second hypothesis, extensive research on parallelized logic programming does not contribute much to DL reasoning. Furthermore, some DL fragments, without disjunction and at-most cardinality restriction constructors, do not profit much from parallelizing non-deterministic branches in tableau expansion.

The research mentioned above is focusing on a shared-memory multi-threading environment. There is also quite some research proposing distributed solutions, and some of these ideas are also worth being tried in a shared-memory environment.

In [21] the idea of applying a *constraint programming* solver, *Mozart*, was proposed to $\mathcal{ALC}$ tableau reasoning in parallel, and the implementation was reported on in [20]. The experimental results show scalability to some extent.

Recently, some research work focuses on how DL reasoning can be applied to *Resource Description Framework (RDF)* and OWL . This trend leads to research on reasoning about massive web ontologies in a scalable way. MapReduce [22], ontology mapping [13], ontology partitioning [10], rule partitioning [24], *distributed hash table (DHT)* [8], swarm intelligence [7], etc., are examples of these approaches. However, we do not consider these approaches as relevant in our context.

5 Conclusion and Future Work

DL has been successfully applied in many domains, amongst which utilizing knowledge on the Internet has become a research focus in recent years. *Scalable* solutions are required for processing an enormous amount of structured information spreading over the Internet. Meanwhile, multi-processor computing facilities are becoming the main stream. Thus, we consider research on parallelism in DL reasoning as necessary to utilize available processing power..

The objective of this research is to explore how parallelism plays a role in tableau-based DL reasoning. A number of tableau-based DL reasoning optimization techniques have been extensively researched, but most of them are investigated in *sequential* contexts, so adapting these methods to the *parallel* context is an important part of this research.

We have partially shown that shared-memory parallel tableau-based DL reasoning can contribute to scalable solutions. This paper introduced our reasoner, Deslog, of which the architecture is devised specially for a shared-memory parallel environment. We presented an aspect of the reasoner's concurrency performance, and a good scalability could be demonstrated for TBox classification.

In the near future, we will enhance Deslog so that it can reason about more expressive DL languages. More powerful tableau expansion rules will be added. Concurrency-suitable tableau optimization techniques will be implemented. Moreover, we plan to investigate non-determinism more closely, and to design corresponding parallel algorithms.

References

1. Aslani, M., Haarslev, V.: Towards parallel classification of TBoxes. In: Proceedings of the 21st International Workshop on Description Logics (DL2008), Dresden, Germany. vol. 353 (2008)
2. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics—first experimental results. In: Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. pp. 485–490 (2010)
3. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: International Joint Conference on Artificial Intelligence. vol. 19, p. 364 (2005)
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: theory, implementation, and applications. Cambridge University Press (2003)

5. Baader, F., Hollunder, B., Nebel, B., Profitlich, H.J., Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. Applied Intelligence 4(2), 109–132 (1994)
6. Bock, J.: Parallel computation techniques for ontology reasoning. In: Proceedings of the 7th International Conference on The Semantic Web. pp. 901–906 (2008)
7. Dentler, K., Guéret, C., Schlobach, S.: Semantic web reasoning by swarm intelligence. In: The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). p. 1 (2009)
8. Fang, Q., Zhao, Y., Yang, G., Zheng, W.: Scalable distributed ontology reasoning using DHT-based partitioning. In: The semantic web: 3rd Asian Semantic Web Conference, ASWC 2008. pp. 91–105 (2008)
9. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: A logical framework for modularity of ontologies. In: Proc. IJCAI. pp. 298–304 (2007)
10. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research 31(1), 273–318 (2008)
11. Grau, B.C., Parsia, B., Sirin, E., Kalyanpur, A.: Modularizing OWL ontologies. In: K-CAP 2005 Workshop on Ontology Management (2005)
12. Haarslev, V., Möller, R., Turhan, A.Y.: Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. Automated Reasoning 2083/2001, 61–75 (2001)
13. Homola, M., Serafini, L.: Towards formal comparison of ontology linking, mapping and importing. In: 23rd International Workshop on Description Logics, DL2010. p. 291 (2010)
14. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web 2(1), 11–21 (2011)
15. Horrocks, I.: Optimising tableaux decision procedures for description logics. Ph.D. thesis, The University of Manchester (1997)
16. Horrocks, I., Sattler, U., Tobies, S.: Reasoning with individuals for the description logic SHIQ. In: Automated deduction-CADE-17: 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000: proceedings. vol. 17, p. 482 (2000)
17. Kazakov, Y.: Consequence-driven reasoning for horn SHIQ ontologies. In: Proceedings of the 21st international jont conference on Artifical intelligence. pp. 2040–2045 (2009)
18. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th International Semantic Web Conference (2011)
19. Liebig, T., Müller, F.: Parallelizing tableaux-based description logic reasoning. In: Proceedings of the 2007 OTM Confederated International Conference on the Move to Meaningful Internet Systems-Volume Part II. pp. 1135–1144 (2007)
20. Meissner, A.: A simple parallel reasoning system for the ALC description logic. In: Computational Collective Intelligence: Semantic Web, Social Networks and Multiagent Systems (First International Conference, ICCCI 2009, Wroclaw, Poland, 2009). pp. 413–424. Springer (2009)
21. Meissner, A., Brzykcy, G.: A parallel deduction for description logics with ALC language. Knowledge-Driven Computing 102, 149–164 (2008)
22. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for EL+. In: 23rd International Workshop on Description Logics DL2010. p. 456 (2010)
23. Simančík, F., Kazakov, Y., Horrocks, I.: Consequence-based reasoning beyond Horn ontologies. In: Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI 2011) (2011)
24. Soma, R., Prasanna, V.K.: Parallel inferencing for OWL knowledge bases. In: Proceedings of the 2008 37th International Conference on Parallel Processing. pp. 75–82 (2008)
25. Wu, J., Haarslev, V.: Planning of axiom absorptions. In: In Proceedings of International Workshop on Description Logics 2008 (2008)