Exploring Parallelization of Conjunctive Branches in Tableau-based Description Logic Reasoning

Kejia Wu and Volker Haarslev

w_kejia@cs.concordia.ca haarslev@cse.concordia.ca
Computer Science, Concordia University, Canada

Abstract. Multiprocessor equipment is cheap and ubiquitous now, but users of description logic (DL) reasoners have to face the awkward fact that the major tableau-based DL reasoners can make use only one of the available processors. Recently, researchers have started investigating how concurrent computing can play a role in tableau-based DL reasoning with the intention of fully exploiting the processing resources of multiprocessor computers. The published research mostly focuses on utilizing disjunctive branches, the *or-part* of tableau expansion trees. We investigated the possibility and the role of concurrently processing conjunctive branches, the *and-part* of tableau expansion trees. In this work, we present an algorithm to process conjunctive branches in parallel and address the key implementation aspects of the algorithm. A research prototype to execute this algorithm has been developed and empirically evaluated. The experimental results are presented and analyzed. We found that parallelizing the processing of conjunctive branches of tableau expansion trees is auspicious and can partly evolve into a scalable solution for DL reasoning.

1 Introduction

Multiprocessor computers are almost everywhere now, such as high-performance computing servers, personal computers, laptops, tablets, phones, watches, etc. In many computing areas it is investigated how to better use the power of multiprocessor computers, and concurrent computing is an important option. Unfortunately, few such advances have been achieved for tableau-based *Description Logic (DL)* reasoning, which is the dominant method adopted by the major DL reasoning systems. None of the major DL reasoning systems can utilize the available processing capacity of multiprocessor computers properly. To be more precise, any of them can only use one processor at a time, no matter how many processors are available on the computer where it is running.

At the same time, DL reasoners are processing more complex ontologies. With the fast development of the Internet, DL reasoning can be considered in the age of *big data*. In the semantic web area, both huge but often simple as well as more and more complicated ontologies are emerging. Even for specialized or highly optimized DL systems it is not easy to reason about these ontologies. So, people are investigating innovative methods to improve reasoning performance. Concurrent reasoning solutions may improve performance for dealing with massive or complicated ontologies.

The wide availability of multiprocessor computing facilities makes concurrent DL reasoning feasible. Several approaches have been reported for concurrent reasoning

in some non-shared or shared memory context [21, 6, 17, 20]. Considering the cost of maintaining communication between computing objects, the latter looks more promising at present. However, compared with the vast research on optimization techniques for tableau-based DL reasoning, there are only few research approaches that employ concurrent computing as an optimization method, and none makes use of processing conjunctive branches of tableau expansion trees.

It is well-known that an expansion tree is an *and-or* tree in tableau-based DL reasoning. Disjunctive branches compose the *or* part of a completion tree, conjunctive branches do the *and* part, and generally the two types of branches interlace with one another. Almost all of the few shared-memory parallelized tableau-based DL reasoning investigations focus on exploiting disjunctive branches in expansion trees.

In the remaining sections, the role of conjunctive branches in tableau expansion trees is addressed, a parallelized tableau-based algorithm which processes conjunctive branches simultaneously is presented, the key points of the algorithm's implementation are discussed, experiments and evaluations on the program are shown and are analyzed, and some related investigations are also mentioned.

2 The Role of Conjunctive Branches

Tableau-based algorithms have been the primary choice of DL reasoning for a long time. The core of a tableau algorithm is a set of rules that are used to construct completion trees. Whether a clash-free completion tree can be built determines the satisfiability of a problem domain. In DL languages with sufficient expressive power, such completion trees are regarded as *and-or* ones [24]. That is to say, both *conjunctive* and *disjunctive* branches exist in the completion trees.

A clash-free completion tree must have at least one disjunctive branch that contains no clashed conjunctive branches. For example, in a skeletal way, a typical tableau algorithm generates the following completion tree at some point when testing the satisfiability of the concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$:

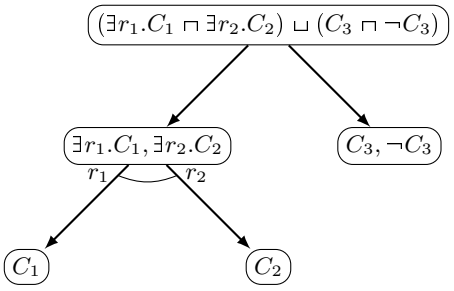

Fig. 1. The tableau expansion tree of testing the satisfiability of $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$.

The concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$ must have the disjunctive branches $\exists r_1.C_1 \sqcap \exists r_2.C_2$ or $C_3 \sqcap \neg C_3$ clash-free only if it is satisfiable. In this case, $C_3 \sqcap \neg C_3 \equiv \bot$, so the satisfiability of $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap \neg C_3)$ is determined by whether $\exists r_1.C_1 \sqcap \exists r_2.C_2$ is satisfiable. That is to say, if the concept $(\exists r_1.C_1 \sqcap \exists r_2.C_2) \sqcup (C_3 \sqcap$

$\neg C_3)$ is satisfiable, both the conjunctive branches $\exists r_1.C_1$ and $\exists r_2.C_2$ must be clash-free. The algorithm has to explore all conjunctive branches unless an unsatisfiability result is entailed.

Testing satisfiability is an essential function in tableau-based DL reasoning, and its goal is to search for a model by expanding concepts descriptions to completion trees, which consist of disjunctive and conjunctive branches. Testing satisfiability is generally used by other DL reasoning services. As we know an important functionality of modern DL systems is classification, which calculates all subsumption relationships entailed by a terminology:

$$\forall C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \forall D^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} : \mathcal{T} \stackrel{?}{\models} C \sqsubseteq D \iff \mathcal{T} \stackrel{?}{\models} \neg C \sqcup D \tag{1}$$

With respect to $\mathcal{T}$, $C \sqsubseteq D$ is proven if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for *every* model $\mathcal{I}$ of $\mathcal{T}$. This is calculated by testing the satisfiability of the concept $C \sqcap \neg D$ with respect to $\mathcal{T}$. The subsumption computation is reducible to testing satisfiability in such a way. It is obvious that $C \not\equiv D$ is the most common case, and thus the majority of such subsumption tests find models. That is to say, in order to gain better performance, a tableau-based DL reasoning algorithm should find models as fast as possible. In a tableau expansion view, such a model is a disjunctive branch with a bundle of conjunctive branches, both clash-free. Considering that such a disjunctive branch exists quite often, faster processing of conjunctive branches in that disjunctive branch should improve reasoning performance. Although research on parallelizing the processing of disjunctive branches is known, parallelizing the processing of conjunctive branches has not been researched so far, but it should play a role in improving the performance of tableau-based DL reasoning.

3 Parallelism

As mentioned in Section 1, many approaches are known that pursue new methods to increase the performance of reasoners, and concurrent computing is an option. In tableau-based DL reasoning, disjunctive and conjunctive branches have always been sequentially processed as of now, although there exists the potential benefit of parallelization.

The search for a model in a disjunctive branch is independent of other disjunctive branches. The satisfiability of a concept is sufficiently supported by any model found among disjunctive branches. With a sequential algorithm, if two disjunctive branches are generated at some point, the second branch is only calculated if the first branch is not clash-free. With a parallel algorithm, multiple disjunctive branches are processed at the same time, and the search terminates when one of them is proven as clash-free. Some research has already been reported on the topic of parallelizing tableau calculation on disjunctive branches [17, 5, 20, 25].

Computation on a conjunctive branch impacts its siblings in a different way than for disjunctive branches. A model is found by a tableau algorithm if and only if all involved conjunctive branch siblings are clash-free. With a sequential algorithm, all conjunctive branches on a disjunctive branch must be explored so that a clash-free *and*-tree can be built. Parallelizing computation on conjunctive branches in a satisfiable context should theoretically improve performance. Given the fact that most satisfiability tests introduced by classification, a key functionality of a DL reasoning system, return positive results, parallelizing conjunctive branches in tableau-based reasoning should play an

important role. However, conjunctive branch parallelization has not been researched as much yet.

Parallelizing the processing of conjunctive branches is necessary to maximally utilize parallel computers. As we discussed in Section 2, the majority of computations of tableau-based DL reasoning find clash-free completion trees, each of which can be considered as a disjunctive branch containing a number of conjunctive branches. According to our experience, subsumption tests in classification are often easily satisfiable. Such a satisfiable disjunctive branch is usually the first one being tested. So, a parallel scheme in that case hardly improves reasoning performance. On the other hand, all conjunctive branches of a clash-free disjunctive branch must be explored and determined as clash-free. Therefore, parallelizing the exploration of potentially clash-free conjunctive branches can improve reasoning performance. Research on parallelizing the processing of conjunctive branches in tableau-based DL reasoning may even play a more important role than on disjunctive branches.

4 Algorithm Design and Implementation

In this section we present a parallel tableau-based DL algorithm. When we use the word *parallel* in the following, a modern shared-memory multi-thread architecture should always be taken into account.

Concurrent algorithms have much more technical features than sequential ones. Some solutions require very tricky techniques. For example, in a sequential context, a DL tableau algorithm terminates the search in a disjunctive branch when a clash is found in a conjunctive branch. Such a termination task needs more complex mechanics in a parallel context. A termination test in a multiple threads context not only needs to check its own state but also the state of its siblings, i.e., it must monitor contradiction detection for all its siblings as the prerequisite for ensuring both *soundness* and *performance*.

The efficient managing of resources is an important tradeoff in designing concurrent algorithms, especially in a shared-memory context. A common pitfall in developing shared-memory parallel algorithms consist of taking unlimited threading for granted, as usually happens in recursive algorithms. A compensation for this flaw is the use of shared data to control resources allocated to a parallel program. However, shared data as well as communication between threads always reduce a parallel program's performance.

Our scheme for controlling continuation resources is using a thread pool, which is normally configured with a fixed size. The members of the pool are reusable, which largely reduces system overhead. The most notable shared data consists of an increasing number of sibling conjunctive branches, and we use a concurrent queue to buffer them. Every threaded reasoning continuation picks a conjunctive branch out of the shared queue and processes it. Also, every continuation has to monitor and report its finding, as mentioned before.

The parallelization of processing conjunctive branches is addressed by Algorithms 1 and 2. It consists of two parts: Algorithm 1 as the control (master) and Algorithm 2 as the continuation (slave). Algorithm 1 first applies tableau expansion rules that are neither a $\exists$-rule nor a $\forall$-rule. The function $clashed?$ returns $true$ if *all* disjunctive branches (i.e., *stages* in $\mathcal{D}\mathfrak{eslog}$ [25]) are not clash-free, otherwise it returns $false$ and

cuts away clashed stages. If all disjunctive branches are not clash-free in this phase, the computation terminates. Otherwise, the model-search is continued on the generated disjunctive branches, which are provisionally clash-free. That is to say, the generating rule produces conjunctive branches which are kept in a buffer. Then the aforementioned thread pool schedules computation continuations on the conjunctive branch buffer. The computation continuation executed by the pooled thread is addressed by Algorithm 2, which is executed by multiple threads simultaneously.

We implemented Algorithms 1 and 2 with our parallelized tableau-based DL reasoning framework $\mathcal{D}\mathfrak{eslog}$ [25]. The underground parallel mechanics of $\mathcal{D}\mathfrak{eslog}$ is supported by the *java.concurrent* package. All working threads processing continuations are activated and pooled in the bootstrap phase. In each subsumption test run, every thread monitors a *volatile* flag that indicates whether a clash has been detected by its siblings and modifies the flag if it finds a clash (Line 2 and 3, Algorithm 2). If a clash has been detected, all threads and the flag are reset.

A performance bottleneck may result from the low level Java concurrency components. For example, we use a concurrent linked queue to buffer immediate conjunctive branches, and the buffer is accessed by a number of threads concurrently. Also, we use *volatile* flags as shared states with the intention of notifying state modification as fast as possible, and the maintenance of the *volatile* variables may require extra processor resources in a shared-memory parallel computing environment. We can design and construct the high level part of the program, but can hardly control the low level facilities on which the program depends.

5 Experiments

Algorithm 1 is expected to improve the performance of tableau-based DL reasoning in such a way that conjunctive branches are processed simultaneously. A higher performance improvement is expected from reasoning about problems where more conjunctive expansion branches are involved. We designed a series of synthesized tests to prove this assumption.

The test cases consist of a set of *Web Ontology Language (OWL)* benchmarks developed on the basis of the *tea* ontology[1]. Their size (and complexity) can be scaled by a *general concept inclusion (GCI)* axiom pattern as follows:

$$\bigsqcup_{i=0}^{\jmath} \forall R_{2i+1}.(C_{2i+1} \sqcap C_{2i+2}) \sqsubseteq \prod_{i=0}^{\jmath} \exists R_{2i}.(C_{2i} \sqcup C_{2i+1}),$$
$$C_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, R_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}, i \in \mathbb{N}, \jmath \in \mathbb{N}, i \leqslant \jmath \quad (2)$$

We defined a set of factors to measure the algorithm's performance. Performance improvement is directly reflected by *thread number* and *speedup*. With the same thread number, reasoning performance varies with the number of involved conjunctive branches. So, our program records the number of involved conjunctive branches, μ, in every satisfiability test. N_μ, the total number of the tests in a set of computations, of which every one processes μ conjunctive branches, is calculated after each run. We discovered that

[1] http://code.google.com/p/deslog/downloads/detail?name=tea.tar.gz

Algorithm 1: $parallelize\text{-}traces(tree, rule\text{-}queue\text{-}without\text{-}\exists\forall, worker\text{-}queue)$

input :
$tree$: a tableau expansion tree.
$rule\text{-}queue\text{-}without\text{-}\exists\forall$: an ordered set containing tableau expansion rules except for $\exists$-rule and $\forall$-rule.
$worker\text{-}queue$: the pool keeping threads.

output :
$some\text{-}trace\text{-}clashed?$: true if clash is found, otherwise false.

```
1  begin
2      reload(rule-queue-without-∃∀);
3      rule ← dequeue(rule-queue-without-∃∀);
4      while rule ≠ ∅ ∧ ¬clashed?(tree) do
5          applicable? ← apply(rule, tree);
6          if applicable? then
7              reload(rule-queue-without-∃∀);
8          end if
9          rule ← dequeue(rule-queue-without-∃∀);
10     end while
11     some-trace-clashed? ← clashed?(tree);
12     if ¬some-trace-clashed? then
13         foreach disjunctive-branch ∈ tree do
14             trace-queue ← generate-trace-queue(disjunctive-branch, rule-∃);
15             some-trace-clashed? ← false;
16             if ¬empty?(trace-queue) then
17                 continue? ←true;
18                 while continue? do
19                     trace ← dequeue(trace-queue);
20                     if trace ≠ ∅ then
21                         worker ← ∅;
22                         while worker = ∅ ∧ ¬some-trace-clashed? do
23                             worker ← get-idle-worker(worker-queue);
24                         end while
25                         if some-trace-clashed? then
26                             continue? ←false;
27                         else
28                             do-job(worker, λ(trace, some-trace-clashed?));
29                         end if
30                     else
31                         if some-trace-clashed? ∨
32                               get-busy-worker(worker-queue) = ∅ then
33                             continue? ←false;
34                         end if
35                     end if
36                 end while
37                 if ¬some-trace-clashed? then
38                     break;
39                 end if
40             end if
41         end foreach
42     end if
43     return some-trace-clashed?;
44 end
```

Algorithm 2: $\lambda(trace, *clashed\text{-}flag?*)$

input :
$trace$: a tableau conjunctive branch.
$*clashed\text{-}flag?*$: a pointer argument indicating whether a clash exists: true if clash is found; otherwise false.

```
1 begin
2 |  apply-tableau-rules(trace);
3 |  *clashed-flag?* ← ¬clash-free?(trace);
4 end
```

the most frequently occurring number of the involved conjunctive branches impacts the final reasoning performance and is the *order* of the conjunctive branches involved in a run. We take the *mean* of the values to calculate this order, which is noted as τ, where at most k conjunctive branches are involved in a satisfiability test:

$$\tau = \frac{\sum_{\mu=0}^{k} N_\mu \times \mu}{\sum_{\mu=0}^{k} N_\mu},\ k \in \mathbb{N},\ 0 \leqslant \tau < +\infty \tag{3}$$

For example, we get $\tau = \frac{5\times0+8\times1+13\times2+21\times3}{5+8+13+21} = \frac{97}{47}$, with respect to the sample data shown in Table 1.

μ	0	1	2	3
N_μ	5	8	13	21

Table 1. A sample data set for Equation 3 with $k = 3$.

We conducted several experiments to evaluate Algorithm 1. According to our knowledge, the hardware environment can have quite an impact on the performance of a parallel program in a shared-memory context [25, 17]. In this case, a 16-core computer running Solaris OS and 64-bit Sun Java 6 was employed to test the program. The 16 processors are manufactured on 2 integrated circuits, each having 8 processors. At most 64G physical memory is accessible by the JVM. With various combinations of the number of processing threads and problem sizes, Algorithm 1 demonstrated the capability of being scaled.

The reasoning performance of Algorithm 1 is illustrated by the results shown in Figures 2-8. When $\tau = 2.09$, i.e., each test processes only $\sim$2 conjunctive branches, parallelizing the processing of conjunctive expansion does not contribute to a performance improvement. It seems that the overhead introduced by threading outclasses the benefits. However, according to our experiments performance improvements can be gained when $\tau \geqslant 3.09$. Better performance improvements come from greater τ values.

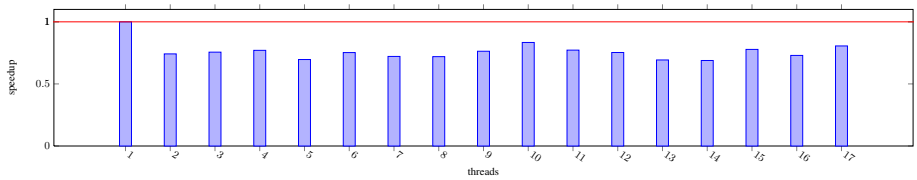

Fig. 2. The speedup when $\jmath = 2$ and $\tau = 2.09$.

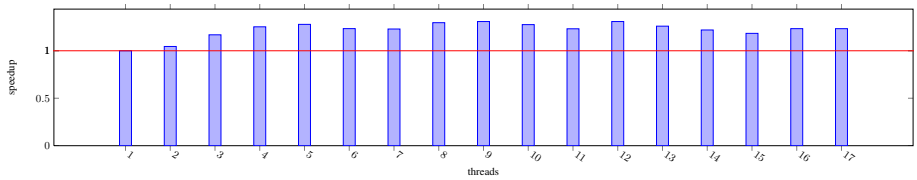

Fig. 3. The speedup when $\jmath = 3$ and $\tau = 3.09$.

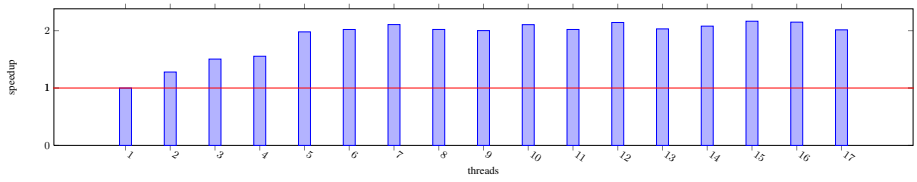

Fig. 4. The speedup when $\jmath = 4$ and $\tau = 5.08$.

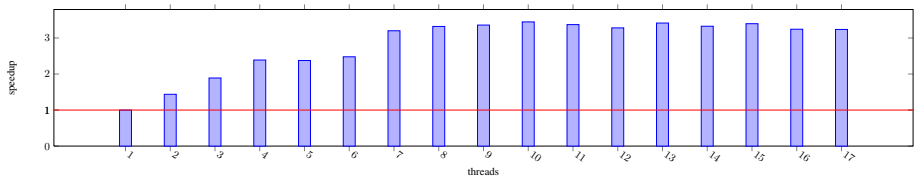

Fig. 5. The speedup when $\jmath = 7$ and $\tau = 7.13$.

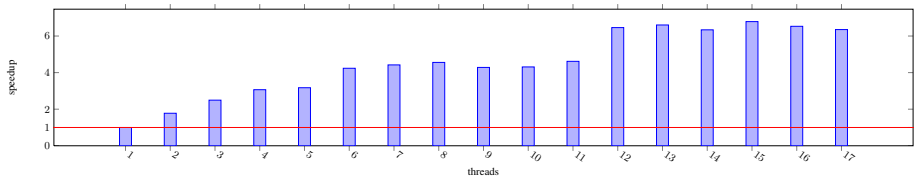

Fig. 6. The speedup when $\jmath = 11$ and $\tau = 12.11$.

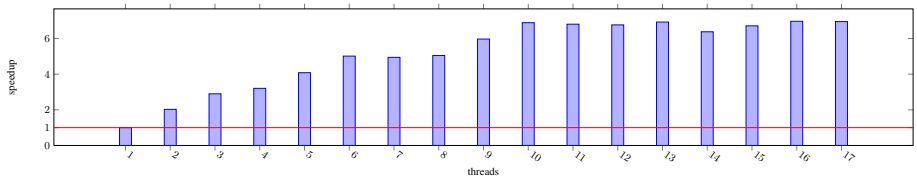

Fig. 7. The speedup when $\jmath = 17$ and $\tau = 18.08$.

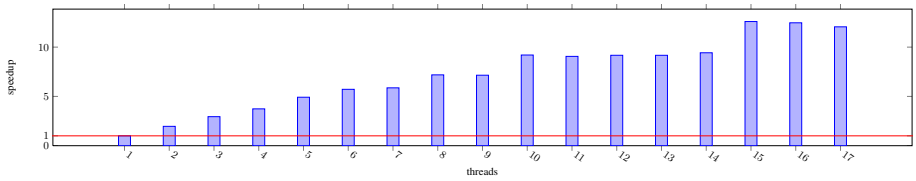

Fig. 8. The speedup when $\jmath = 28$ and $\tau = 29.06$.

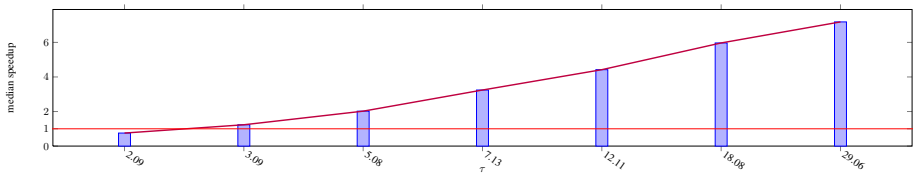

Fig. 9. The median speedup trend of the variety of τ values.

The scalability of parallelizing the processing of conjunctive branches is summarized in Figure 9 by illustrating the speedup trend, which is based on the median speedup values from our 9-thread tests (Figure 2-8), given the observed τ values.

Besides synthesized test cases, we also tested a real-world ontology, *fly_anatomy*. Figure 10 shows the result for *fly_anatomy*. The maximum speedup value is around the value of τ, which is in this test case 2.91, and we see that the peak value of the speedup is greater than 2 and that there exists a linear speedup increase before reaching the peak value. This test result shows a stable scalability to some degree.

We amend Equation 3 to a more general form as indicated by Equation 4, in order to illustrate the program's impacts on real-world ontologies:

$$\tau_q = \frac{\sum_{\mu=q}^{k} N_\mu \times \mu}{\sum_{\mu=q}^{k} N_\mu},\ 0 \leqslant q \leqslant k,\ k \in \mathbb{N},\ 0 \leqslant \tau < +\infty \tag{4}$$

Here, q is a lower bound of a sample space. Namely, tests with conjunctive branches less than q are bypassed. With Equation 4, we can focus on the tests with greater q value, e.g. $q = 3$. Figure 11 shows the speedups gained by testing some ontologies, with $q = 4$.

Scalability is the most interesting point in this research. Optimistically, we expect to gain linear or even super-linear scalability.[2] In the circumstances of expecting minimal overheads, we anticipate the ratio, between speed-up and the number of thread, $e \geqslant 1$ in accordance with $e = \frac{s}{n},\ n \leqslant \tau, n \leqslant p$, where s is speedup, n is thread number, and p is the total number of processors. However, $e \leqslant 1$ is the most normal case in practice. According to our tests, $e = 0.8398$ is the greatest value, which occurs when

[2] A super-linear speedup is controversial but sometimes observed, and we accept it as the result of the combined effectiveness of hardware, software, and algorithms [23, 11, 10, 22, 1].

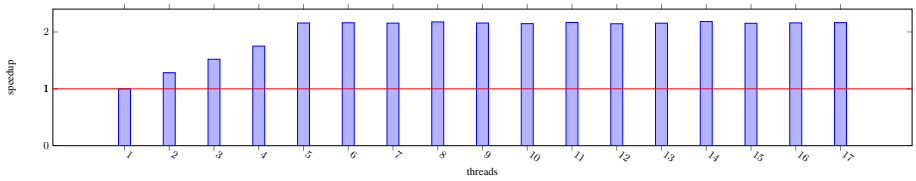

Fig. 10. Test ontology fly_anatomy, $\tau = 2.91$.

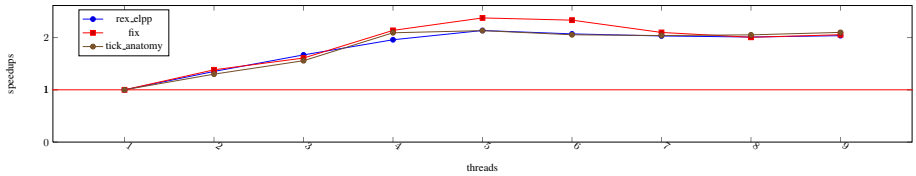

Fig. 11. Test ontologies rex_elpp, fix, and tick_anatomy with $\tau_4 = 4.73$, 5.00, and 4.22 respectively.

$s = 12.5971$, $n = 15$, $\tau = 29.06$, and $p = 16$ (see Figure 8). It is obvious that a certain system overhead cannot be avoided and must hinder the program from reaching the normal peak value. Furthermore, most tableau-based satisfiability tests for classification find models, as mentioned above, and parallelizing the processing of conjunctive branches is useful in the case of satisfiable tests. However, negative tests entailing unsatisfiability predominantly exist in classification, too. If a number of clashes can be detected before processing a bundle of conjunctive branches, this parallel algorithm can hardly contribute to a performance gain.

6 Related Work

In the past, tableau-based DL reasoning techniques have been investigated extensively. In [24] a tableau algorithm was proposed to reason about $\mathcal{ALC}$. After that, tableau algorithms have been presented along with the growing expressivity of DL languages [18, 15, 4, 14, 8, 7, 9]. The core of these algorithms is searching for models by building completion trees, which are *and-or* trees.

Generic tableau algorithms generally run with a low performance, so a number of optimization techniques have been developed for real-world reasoners. Without some of these optimization techniques, a tableau-based DL reasoning system remains unusable. Examples of important optimization techniques are dependency-directed backjumping, lazy unfolding, axiom absorption, etc [12, 13]. However, most of these optimization techniques do not improve reasoning performance by making use of parallelized processing. None of the well-known tableau-based DL reasoning systems has necessarily needed more than one processor in this multiprocessor computing age so far.

For the purpose of exploiting multiprocessor computers' power to deal with increasingly complex ontologies, researchers have started looking for scalable solutions, among which concurrent computing plays a role. A variety of concurrency-oriented DL reasoning research has been presented recently. [17] reported on a parallel $\mathcal{SHN}$ reasoner, which can execute *disjunction* and *at-most cardinality restriction* rules in parallel,

as well as some primary DL tableau optimization techniques. [2, 3] presented the first algorithm for parallel TBox classification achieving (super)linear scalability. [16] reported on a reasoner that can classify $\mathcal{EL}$ ontologies *concurrently*, and its *consequence-based* parallel algorithm brought encouraging results for non-tableau-based reasoning. [25] presented an $\mathcal{ALC}$ reasoner that can process disjunctive branches in parallel. [20, 19] proposed the idea of applying a constraint programming solver to DL reasoning. Most of these studies focus on how to best make use of disjunctive branches, the *or*-part of tableau expansion trees. According to the best of our knowledge, there is no reported research on investigating the role of simultaneously processing conjunctive branches, the *and*-part of tableau expansion trees. Our contribution is to initiate research on conjunction parallelization in tableau-based DL reasoning.

7 Summary and Future Work

Cheap multiprocessor computing resources are ubiquitous now. How these powerful computers can be used adequately is an important topic in computing areas, and some research, such as graphics processing, has progressed a lot. Unfortunately, DL reasoning just started its journey in this direction, and any full-scale DL/OWL reasoner can only use one single processor at a time no matter how many are available. At the same time, ontologies processed by DL reasoning systems are becoming bigger and more complex because of the fast development of the semantic web technology. A possible solution is the integration of concurrent computing, or more specifically, concurrent reasoning, which should make full use of the availability of multiprocessor computing resources and may improve performance in a scalable way. Our research proved that such scalability is possible.

We have shown that the computing performance of tableau-based DL reasoning can be improved by parallelizing the processing of conjunctive branches of expansion trees. All of the investigations that explore parallelized tableau-based DL reasoning make an effort to exploit simultaneous processing of disjunctive branches in tableau expansion trees. On the other hand, our research is the first one to seriously investigate the parallel processing of conjunctive branches in tableau expansion trees. We addressed the role of conjunctive branches in tableau expansion trees and noticeable points of parallelizing the processing of conjunctive branches. We presented a parallel algorithm that simultaneously calculates conjunctive branches. We discussed the key characteristics of implementing the algorithm. We evaluated the program, and the essential effectiveness of the algorithm was shown by synthesized tests. We analyzed the scalability of the algorithm based on our proposed τ metric.

In the future, more efficient memory management techniques may be explored. At present, we implemented the algorithm with Java. When processing complex ontologies, which generally need relatively large memory sizes, Java's memory management, e.g., garbage collection, has a high impact on reasoning performance. Such influence is generally negative and could be avoided by using C++ for implementation. Moreover, thread scheduling used by the present implementation heavily depends on the Java concurrency package. More elaborate designs may improve performance further.

References

1. Alba, E.: Parallel evolutionary algorithms can achieve super-linear performance. Information Processing Letters 82(1), 7–13 (2002)
2. Aslani, M., Haarslev, V.: Parallel TBox classification in description logics—first experimental results. In: Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. pp. 485–490 (2010)
3. Aslani, M., Haarslev, V.: Concurrent classification of OWL ontologies—an empirical evaluation. In: Proceedings of the 2012 International Workshop on Description Logics. pp. 7–10 (2012)
4. Baader, F., Brandt, S., Lutz, C.: Pushing the EL envelope. In: 2005 International Joint Conference on Artificial Intelligence. vol. 19, p. 364 (2005)
5. Bock, J.: Parallel computation techniques for ontology reasoning. In: Proceedings of the 7th International Conference on The Semantic Web. pp. 901–906 (2008)
6. 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)
7. Ding, Y.: Tableau-based reasoning for description logics with inverse roles and number restrictions. Ph.D. thesis, Concordia University (2008)
8. Ding, Y., Haarslev, V.: An exptime tableau-based decision procedure for ALCQI. In: Proceedings of the 2007 International Workshop on Description Logics (2007)
9. Ding, Y., Haarslev, V.: A procedure for description logic ALCFI. Journal of Logic and Computation 20(1), 5–34 (2010)
10. Donaldson, V., Berman, F., Paturi, R.: Program speedup in a heterogeneous computing network. Journal of Parallel and Distributed Computing 21(3), 316–322 (1994)
11. Helmbold, D.P., McDowell, C.E.: Modelling speedup (n) greater than n. IEEE Transactions on Parallel and Distributed Systems 1(2), 250–256 (1990)
12. Horrocks, I.: Optimising tableaux decision procedures for description logics. Ph.D. thesis, The University of Manchester (1997)
13. Horrocks, I.: Implementation and optimization techniques. In: Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.) The description logic handbook: theory, implementation, and applications, chap. 9. Cambridge University Press (2003)
14. Horrocks, I., Sattler, U.: A tableau decision procedure for SHOIQ. Journal of Automated Reasoning 39, 249–276 (2007)
15. 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 2000, Proceedings. p. 482 (2000)
16. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th International Semantic Web Conference (2011)
17. 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)
18. Lipkis, T.: A KL-ONE classifier. In: Proceedings of the 1981 KL-ONE Workshop. pp. 128–145 (1982)
19. 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, October 2009). pp. 413–424 (2009)
20. Meissner, A., Brzykcy, G.: A parallel deduction for description logics with ALC language. Knowledge-Driven Computing 102, 149–164 (2008)

21. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce algorithm for EL+. In: Proceedings of the 2010 International Workshop on Description Logics. p. 456 (2010)
22. Nagashima, U., Hyugaji, S., Sekiguchi, S., Sato, M., Hosoya, H.: An experience with super-linear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes. Parallel computing 21(9), 1491–1504 (1995)
23. Quinn, M.J., Deo, N.: An upper bound for the speedup of parallel best-bound branch-and-bound algorithms. BIT Numerical Mathematics 26(1), 35–43 (1986)
24. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence 1(48), 1–26 (1991)
25. Wu, K., Haarslev, V.: A parallel reasoner for the description logic ALC. In: Proceedings of the 2012 International Workshop on Description Logics (2012)