A Framework for Parallelizing OWL Classification in Description Logic Reasoners

Zixi Quan

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montreal, Quebec, Canada

March 2019

© Zixi Quan, 2019

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Zixi Quan

Entitled: A Framework for Parallelizing OWL Classification in Description Logic Reasoners

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

Dr. S. Samuel Li

______________________________ External Examiner

Dr. Weichang Du

______________________________ External to Program

Dr. Chun Wang

______________________________ Examiner

Dr. Dhrubajyoti Goswami

______________________________ Examiner

Dr. Yuhong Yan

______________________________ Thesis Supervisor

Dr. Volker Haarslev

Approved by ______________________________

Dr. Volker Haarslev, Graduate Program Director

April 23, 2019 ______________________________

Dr. Amir Asif, Dean

Gina Cody School of Engineering & Computer Science

ABSTRACT

A Framework for Parallelizing OWL Classification in Description Logic Reasoners

Zixi Quan, Ph.D.

Concordia University, 2019

The *Web Ontology Language* (OWL) is a widely used knowledge representation language for describing knowledge in application domains by using classes, properties, and individuals. Ontology classification is an important and widely used service that computes a taxonomy of all classes occurring in an ontology. It can require significant amounts of runtime, but most OWL reasoners do not support any kind of parallel processing.

This thesis reports on a black-box approach to parallelize existing description logic (DL) reasoners for the Web Ontology Language. We focus on OWL ontology classification, which is an important inference service and supported by every major OWL/DL reasoner. To the best of our knowledge, we are the first to propose a flexible parallel framework which can be applied to existing OWL reasoners in order to speed up their classification process. There are two versions of our methods discussed: (i) the first version implements a novel thread-level parallel architecture with two parallel strategies to achieve a good speedup factor with an increasing number of threads, but does not rely on locking techniques and thus avoids possible race conditions. (ii) The improved version implements an improved data structure and various parallel computing techniques for precomputing and classification to reduce the overhead of processing ontologies and compete with other DL reasoners based on the wall clock time for classification.

In order to test the performance of both versions of our approaches, we use a real-world repository for choosing the tested ontologies. For the first version of our

approach, we evaluated our prototype implementation with a set of selected real-world ontologies. Our experiments demonstrate a very good scalability resulting in a speedup that is linear to the number of available cores. For the second version, its performance is evaluated by parallelizing major OWL reasoners for concept classification. Currently, we mainly focus on comparison with two popular DL reasoners: Hermit and JFact. In comparison to the selected black-box reasoners, our results demonstrate that the wall clock time of ontology classification can be improved by one order of magnitude for most real-world ontologies in the repository.

Acknowledgements

When writing this thesis, it feels like that just I finished an extraordinary adventure. I started this adventure for my curiosity of knowledge exploration. There are always many options in front of you and you have to decide when and where you should arrive this place and leave for your next destinations. Until one day, you look back and find how grateful and lucky you are during this journey.

First and foremost, I would like to express my heartfelt appreciations to my supervisor Dr. Volker Haarslev. He gave me this precious opportunity to start this exploration four and half years ago and guided me into the right directions whenever I got lost. I cannot remember how many times he said to me "Congratulations! You survived". His encouragement, constant support and constructive feedback can always help me conquer the difficulties and lead me to the next destination. This research cannot be done without his extensive expertise in this domain and insightful visions.

Secondly, I would like to thank my committee members, who gave me their positive support and feedback for each step during my PhD. I started my Ph.D. by attending Dr. Dhrubajyoti Goswami's course, which became the foundation of this research. Dr. Chun Wang and Dr. Yuhong Yan used their plentiful knowledge to provide me useful suggestions.

Thirdly, I am very appreciative to work with Dr. Nancy Acemian for more than 4 years. She used her expertise in teaching to give me valuable opinions and I gained plentiful teaching experience. The gratitude of her constant moral support during my Ph.D. is beyond my words.

Furthermore, to my friends and colleagues, they dedicate their time and energy to discuss the problems I met and we share all the memorable times together. I feel

exceptionally lucky and thankful to have them during my Ph.D. and it becomes a precious memory in my life.

Last but not the least, I would like to delicate this thesis to my parents and extended family. Thanks for their encouragement to make me have the courage and confidence to start this journey. Their unconditional love and support gives me the best backup and comfort for every step. I am heartfelt thankful to have them in my life.

And thank you to start reading this thesis.

Contents

List of Figures xi

List of Tables xiii

List of Abbreviations xv

1 Introduction 1

1.1 Thesis Objectives . . . 3

1.2 Thesis Contributions . . . 4

1.3 Thesis Outline . . . 6

2 Preliminaries 8

2.1 Introduction . . . 8

2.2 Description Logics . . . 8

2.2.1 Description Language $\mathcal{ALC}$ . . . 8

2.2.2 Tableau Algorithm . . . 13

2.2.3 Transitive Closure . . . 14

2.3 Reasoning . . . 14

2.3.1 RDF Reasoning . . . 15

2.3.2 Resolution-based Reasoning . . . 15

2.3.3 Tableau-based Reasoning . . . 16

2.4 Reasoning Systems . . . 16
2.4.1 Sequential Systems . . . 16
2.4.2 Concurrent Systems . . . 17
2.4.3 Distributed Systems . . . 17
2.5 Parallel Computing . . . 18
2.5.1 High-performance Computing . . . 18
2.5.2 Atomic Data . . . 19
2.5.3 Work-Stealing . . . 19
2.5.4 Hyper-Threading . . . 20

3 Background and Related Work **22**
3.1 Introduction . . . 22
3.2 Sequential Classification Methods . . . 22
3.2.1 Brute Force . . . 23
3.2.2 Simple Traversal Method . . . 24
3.2.3 Enhanced Traversal Method . . . 27
3.2.4 Optimized Classification Method . . . 30
3.3 Parallel Classification and Reasoning Methods . . . 32
3.3.1 Parallel TBox Classification Algorithm . . . 33
3.3.2 Merge Classification . . . 37
3.3.3 Scalable and Parallel Reasoning Approach . . . 40
3.3.4 Distributed Reasoning Architecture . . . 41
3.4 Parallel Reasoning Techniques . . . 42
3.4.1 MapReduce . . . 42
3.4.2 ELK . . . 44
3.4.3 Snorocket . . . 47
3.4.4 Konclude . . . 48

3.5 Summary 50

4 Parallel Reasoning **52**

4.1 Introduction 52

4.2 Architecture 53

4.3 Ontology Classification 55

4.3.1 Random Division Strategy 56

4.3.2 Group Division Strategy 58

4.3.3 Ontology Taxonomy 60

4.4 Optimization 64

4.4.1 Half-Matrix Structure 64

4.4.2 Improved Division Strategy 65

4.4.3 Optimized Parallel Phase 67

4.5 Summary 74

5 Improved Parallel Classification **75**

5.1 Introduction 75

5.2 Improved Data Structure 76

5.2.1 Atomic Half-Matrix Structure $\mathcal{F}$ 76

5.2.2 Maintaining Sets 80

5.3 Improved Ontology Classification 87

5.3.1 Precomputing Phase 87

5.3.2 Classification Phase 89

5.4 Summary 91

6 Evaluation **92**

6.1 First Evaluation 92

6.1.1 Benchmarks 92
6.1.2 Ontology Scale 93
6.1.3 Ontology Complexity 95
6.1.4 Load Balancing 99
6.1.5 Summary 101
6.2 Improved Parallel Classification 101
6.2.1 Benchmarks 102
6.2.2 Precomputing Phase 103
6.2.3 Improved Classification 105
6.2.4 Load Banlancing 107
6.2.5 Comparison with DL Reasoners 109
6.2.6 Summary 109

7 Conclusion **112**
7.1 Thesis Contributions 112
7.2 Future Work 114

Bibliography **116**

Appendix **124**

List of Figures

3.1 Insert concept c through top search of simple traversal method 25
3.2 Inserted concept c is a direct successor of Y 29
3.3 Concept c is not a direct successor in the hierarchy 29
3.4 Possible Relationships: nodes represent classes, solid edges represent pairs in K, and the light dashed line represents a pair that can be in P only if the pair represented by the dashed line is in P or K. (Adapted from [21]) 30
3.5 Final complete subsumption hierarchy (adapted from [3]) 35
3.6 The given TBox and the classified terminology hierarchy 38
3.7 The subsumption hierarchy divisions of different groups (left two) and the subsumption hierarchy after merging both (right) 39
3.8 Main modules of ELK and information flow during classification (adapted from [30]) 45

4.1 The Architecture of Parallel TBox Classification Approach 55
4.2 Scheduling results for Example 3.2 60
4.3 Partial Hierarchy for the concepts in different threads 63
4.4 The whole concept hierarchy of $\mathcal{O}$ 63
4.5 Improved scheduling results for Example 4.2 66
4.6 An Example for Situation 2.3.1 and 2.3.2 68

4.7 Counter examples for 'delete all concepts $X \in K_A$ from P_B' 69
4.8 Counter Examples for Situation 2.4 70
4.9 More Counter Examples for Situation 2.4 70

5.1 Initialization of half-matrix 77
5.2 Complete changes of $\mathcal{F}$ after applying rules 86
5.3 Using atomic operations to solve conflicts in $\mathcal{A}_1$ 87
5.4 Parallel precomputing phase 88
5.5 *Work-Stealing* strategy applied between T_1 and T_3 89

6.1 Speedup factors for ontologies from Table 6.1 with an increasing number of concepts (n = number of concepts) 94
6.2 Speedup factors for ontologies with QCRs from Table 6.4 (q = number of QCRs) 98
6.3 Division cycle result of *ncitations_functional.owl* (concepts = 2332, threads = 10, random division cycle = 10, group division cycle = 1) 100

List of Tables

2.1 Syntax and semantics of descriptions in $\mathcal{ALC}$ 9

2.2 The completion rules for $\mathcal{ALCH}$ 10

3.1 The completion rules for $\mathcal{EL}_+$ and keys for applying MapReduce 43

6.1 Metrics of tested OWL ontologies 93

6.2 Metrics of the used OWL ontologies with QCRs 96

6.3 Time metrics using 10 workers (in milliseconds) (Ave = Average, Med = Median, Dev = Deviation) 97

6.4 Metrics of tested ontologies for precomputing (Equi = Equivalence Axioms, Disjoint = Disjointness Axioms) 103

6.5 Precomputing Results using Hermit (timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds, T = number of threads, >1,000 = more than 1000 times) 104

6.6 Time Metrics of tested OWL ontologies (timeout (TO) = 1,000 seconds, Dev = Deviation, Med = Median, Ave = Average) 105

6.7 Improved classification results using Hermit (timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds) 106

6.8 Improved classification results using Hermit (timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds, P(ara) = Parallel, W = without work stealing) 108

6.9 Time Metrics of tested OWL ontologies using parallel framework with Hermit (timeout (TO) = 1000 seconds) (Sequ = Sequential, Para = Parallel) 110
6.10 Time Metrics of tested OWL ontologies using parallel framework with Hermit (timeout (TO) = 1000 seconds) (Sequ = Sequential, Para = Parallel) 111

List of Abbreviations

KR	**K**nowledge **R**epresentation
OWL	**W**eb **O**ntology **L**anguage
DL	**D**escription **L**ogic
AI	**A**rtificial **I**ntelligence
ALC	**A**ttributive **C**oncept **L**anguage
GCI	**G**eneral **C**oncept **I**nclusion Axioms
TBox	**T**erminological Axioms
ABox	**A**ssertional Axioms
KB	**K**nowledge **B**ase
QCR	**Q**ualified **C**ardinality **R**estriction
RDF	**R**esource **D**escription **F**ramework
RDFS	**R**esource **D**escription **F**ramework **S**chema
URI	**U**niform **R**esource **I**dentifier
FOL	**F**irst **O**rder **L**ogic
HPC	**H**igh-**P**erformance **C**omputing
HT	**H**yper-**T**hreading
D & G	**D**ivide-and-**C**onquer Algorithm
WCT	**W**all **C**lock **T**ime
PW	**P**arallel **W**ithout work-stealing
SMP	**S**ymmetric **M**ulti-**P**rocessing

1 Introduction

The Web Ontology Language (OWL) as part of the semantic web [8] is a widely used knowledge representation language for describing knowledge in application domains. A major topic of knowledge representation focuses on representing information in a form that computer systems can utilize to solve complex problems. The selected knowledge representation formalism is descriptions logics (DLs) [4], which is a family of formal knowledge representation languages. It is used to describe and reason about relevant concepts (terminological knowledge - TBox) and individuals (assertional knowledge - ABox) of a particular application domain. The widely used Web Ontology Language (OWL) is based on DLs. One of the reasoning components in DL systems is an engine known as classifier which infers entailed subsumption relations from knowledge bases. Research for most DL reasoners is focused on optimizing classification using one single processing core [7, 24, 16]. Considering the ubiquitous availability of multi-processor and multi-core processing units not many OWL reasoners can perform inference services concurrently or in parallel.

In the past various parallel reasoning methods have been proposed: A distributed reasoning architecture to accomplish reasoning through a combination of multiple ontologies interconnected by semantic mappings [52]; A research methodology for scalable reasoning using multiple computational resources [57]; A parallel TBox classification approach to build subsumption hierarchies [3]; An optimized consequence-based procedure using multiple cores/processors for classification of ontologies expressed in

the tractable $\mathcal{EL}$ fragment of OWL [29]; Meissner [36] applied some computation rules in a simple parallel reasoning system; A parallel DL reasoner for $\mathcal{ALC}$ [60, 61]; Merge-based parallel OWL classification [62]; A rule-based distributed reasoning framework that can support any given rule set [42]; a framework to formalize the decision problems on parallel correctness and transfer of parallel correctness, providing semantical characterizations, and obtaining tight complexity bounds [2].

High performance computing (HPC) methods can offer a scalable solution to speed up OWL reasoning. Compared with sequential OWL reasoners, such as Racer [25], FaCT++ [55], and HermiT [22], parallel OWL reasoners work concurrently and distribute the whole task into smaller subparts to speed up the process. A few OWL reasoners integrated parallelization techniques; Konclude [53] is highly efficient but its TBox classification is sequential; ELK [29] supports parallel TBox classification but is restricted to the very small $\mathcal{EL}$ fragment of OWL. Moreover, some other parallel DL reasoning methods have shown promising results in the past few years such as the first parallel approach for TBox classification [3] using a shared-tree data structure, merge classification [60, 61, 62] implementing parallel divide-and-conquer approaches, and [19] proposing a parallel framework for handling non-determinism caused by qualified cardinality restrictions.

This work is motivated by previous parallel approaches and also expands ideas about applying parallel computing techniques to DL reasoning. First, considering the variety and differences of parallelism, thread-level parallelism is suitable for the system which requires constant exchange of information and parallel execution in the meanwhile. Second, in order to reduce the runtime during processing, we need to design a data structure which can avoid the use of locks as much as possible for multi-processor and multi-thread systems. Third, when it comes to the problem of scalability, it is also important for us to speed up the whole process of ontology classification

and balance the load of all available processors. Finally, although many parallel approaches and sequential DL reasoners have been developed, there are no consistent and widely accepted solutions to solve the problem of parallel classification for various kinds of ontologies.

Taking all the above questions into account, we propose a general parallel reasoning framework which can be used to parallelize the classification process of OWL reasoners. This approach is implemented with a shared-memory architecture to exchange information among different threads, atomic global data structures to avoid locks during classification, and various new strategies, such as work-stealing and hyper-threading designed for parallel subsumption testing to speedup the whole process of ontology classification. In order to keep the architecture universal we choose existing OWL/DL reasoners as black-box reasoners for deciding satisfiability and subsumption.

1.1 Thesis Objectives

This research is mainly focused on design and implementation of a black-box approach of OWL ontology classification to parallelize existing DL reasoners. The main objectives of a parallel framework which can be applied to existing DL reasoners are as follows:

- Flexible architecture: The existing DL reasoners are selected as a black-box reasoner to test the satisfiability and subsumption relations of concepts. A flexible architecture is necessary to make sure that this framework can be applied to different reasoners.
- Lock-free data structure: Considering the problems of locks, which can affect the experimental results by increasing the waiting time of exchanging and updating

information, we decide to design a lock-free data structure which can not only ensure data consistency but also avoid conflicts among multiple processes.

- Parallel computing techniques: Given that various parallel techniques could be applied to improve the performance of classification, it is important to choose and design the strategies that can be adapted to the flexible framework and speedup the whole classification process.

- Soundness and completeness: Since this algorithm is implemented with a black-box reasoner, the soundness can be guaranteed if the algorithms and the selected black-box reasoner are sound. In addition, every subsumption test between each pair of satisfiable concepts is derived properly and correctly to ensure completeness.

- Scalability: The parallel classification methods can outperform sequential reasoners on various different ontologies categorized by complexity and particularly scalability. A speedup factor can be achieved by increasing the number of available threads and a shorter runtime to compete with sequential reasoners.

- Load Balancing: In order to balance variations in partitions and subsumption tests of different OWL reasoners, different parallel strategies are considered and designed to schedule each processor in an organized way to speed up the wall clock time of ontology classification.

1.2 Thesis Contributions

Our research has received the attention from the DL community as well as the parallel computing community for its contribution to utilizing parallel computing techniques for ontology classification. This work is considered as a novel application domain

to achieve better performance competing with sequential DL reasoners. The main contributions are described as follows:

- The first contribution is the design of a flexible parallel framework which can be used for existing sequential reasoners to speedup the classification process. The first version of this approach (see Chapter 4) has been published in [46] and [47] which presented a thread-level parallel architecture for ontology classification and ideally suited for shared-memory SMP servers, but does not rely on locking techniques and thus avoids possible race conditions. The improved version of the new approach [48] (see Chapter 5) can be applied to existing OWL reasoners and speed up their classification process.

- The second contribution belongs to three different parallel strategies. In the first version, random division and group division strategies (see Section 4.3) are designed and used to reduce the total number of concepts to be classified. For the construction of ontology taxonomy, a divide-and-conquer algorithm is implemented to compute partial hierarchies and update the whole hierarchy in parallel. In the improved version, an enhanced work-stealing strategy (see Section 5.3.2) is designed and applied. This strategy not only reduces the overhead but also reschedules all the available resources during processing, which results in a shorter runtime when compared with black-box reasoners.

- The third contribution relies on a novel atomic half-matrix data structure, which is lock-free and can ensure the completeness of the approach. The first version structure (see Section 4.4.1) consists of a possible list and remaining list for all the satisfiable concepts of an ontology to record all the subsumption relations. In the second version, the data structure is extended to record subsumee, equivalent

and disjoint sets (see Section 5.2), which results in more relations that can be inferred without subsumption testing.

- The fourth contribution is on optimization techniques implemented by applying transitive closure and parallel precomputing. In the first version, according to the transitive closure (see Section 2.2.3), plenty of subsumption relations among concepts are inferred without testing (see Section 4.4.3). In the second version, because of applying parallelism to precomputing, more relations are found by using OWL API [26] to retrieve all declared axioms of an ontology without subsumption tests (see Section 5.3.1).

- The last contribution focuses on the performance of both versions of the methods (see Chapter 6). The first prototype is evaluated with a set of real-world ontologies (see Section 6.1). The results demonstrate a very good scalability resulting in a speedup that is linear to the number of available cores. For the improved version (see Section 6.2), in comparison to the selected black-box reasoner, the results demonstrate that the wall clock time of ontology classification can be improved by one order of magnitude for most real-world ontologies.

1.3 Thesis Outline

The following chapters of this thesis is outlined as follows:

- Chapter 2 (Preliminaries) introduces the basic knowledge of description logic and its inference services, reasoning systems, and the current popular computing techniques.

- Chapter 3 (Background and Related Work) gives a brief review of previous related research using different parallel techniques on classification and reasoning and their performance.

- Chapter 4 (Parallel Reasoning) presents the first version of this framework, which includes two different parallel strategies and a flexible framework for TBox classification.

- Chapter 5 (Improved Parallel Classification) states the improved version of this research, including improved half-matrix data structure, precomputing and classification phases and improved work-stealing strategy to achieve a better performance.

- Chapter 6 (Evaluation) illustrates the results of the two different versions and also explains the impact factors, such as scalability, complexity of ontologies and the improvements of speedup factors and load balancing.

- Chapter 7 (Conclusion) concludes both the theoretical and practical contributions of this work and proposes some potential future work which could be studied for further research.

2 Preliminaries

2.1 Introduction

In this chapter, relevant background is introduced. It is divided into four parts. The first part is about the basics of description logic as well as tableau algorithms. The reasoning methods and systems are introduced in the second and third part. In the last part, there are some relevant parallel computing techniques are presented.

2.2 Description Logics

A major topic of knowledge representation (KR) focuses on representing information in a form that computer systems can utilize to solve complex problems. The selected knowledge representation formalism is descriptions logics (DLs) [4], which is a family of formal knowledge representation languages. It is used to describe and reason about relevant concepts (terminological knowledge - TBox) and individuals (assertional knowledge) of a particular application domain. The widely used Web Ontology Language (OWL) is based on DLs.

2.2.1 Description Language $\mathcal{ALC}$

Syntax and Semantics

The Description Logic Attributive Concept Description Language ($\mathcal{ALC}$) proposed by Schmidt-Schauß and Smolka [50] was the first DL where a complete reasoning algorithm was provided. To formally define an $\mathcal{ALC}$ knowledge base, we denote with N_C a set of concept names of domain elements with common characteristics, N_R a set of role names with a binary relationship between domain elements, and N_O a set of individual names within the represented domain.

TABLE 2.1: Syntax and semantics of descriptions in $\mathcal{ALC}$

Syntax	Semantics
$\top$	$\Delta^{\mathcal{I}}$
$\bot$	$\emptyset$
$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$

The formal definition of the semantics of $\mathcal{ALC}$ is given by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, consisting of a non-empty set $\Delta^{\mathcal{I}}$ called domain and an interpretation function $\cdot^{\mathcal{I}}$. The interpretation function $\cdot^{\mathcal{I}}$ maps every individual a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, every concept A to $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and every role R to $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The description of syntax and semantics of $\mathcal{ALC}$ concept expressions is shown in Table 2.1, where $C, D \in N_C$ are arbitrary concepts and $R \in N_R$ is a role.

Satisfiability

A concept C is satisfiable if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$, i.e., there

exists an individual $x \in \Delta^{\mathcal{I}}$ which is an instance of C, $x \in C^{\mathcal{I}}$. Otherwise, the concept C is unsatisfiable.

TBox

Terminological axioms include role inclusion axioms, which have the form $R \sqsubseteq S$ where $R, S \in N_R$, and general concept inclusion axioms (GCI), which have the form $C \sqsubseteq D$ where C, D are concept expressions. A TBox consists of a finite set of terminological axioms. A TBox $\mathcal{T}$ is satisfiable if there exists an interpretation $\mathcal{I}$ that satisfies all the axioms in $\mathcal{T}$, i.e., for every axiom $C \sqsubseteq D$ ($R \sqsubseteq S$) $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ ($R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$) must hold. Such an interpretation $\mathcal{I}$ is called a model of $\mathcal{T}$ and $\mathcal{T}$ is called consistent. A concept equivalence axiom of the form $C \equiv D$ is an abbreviation for the axioms $C \sqsubseteq D$ and $D \sqsubseteq C$.

TABLE 2.2: The completion rules for $\mathcal{ALCH}$

$\sqcap$-Rule	**If** $C \sqcap D \in L(v)$ and $\{C, D\} \not\subseteq L(v)$ **then** add C and D to $L(v)$
$\sqcup$-Rule	**If** $C \sqcup D \in L(v)$ and $\{C, D\} \cap L(v) = \emptyset$ **then** add X to $L(v)$ with X chosen from $\{C, D\}$
$\forall$-Rule	**If** $R \in L(\langle v, v' \rangle)$, $\forall R.C \in L(v)$ and $C \notin L(v')$ **then** add C to $L(v')$
$\exists$-Rule	**If** $\exists R.C \in L(v)$, no v' exists with $R \in L(\langle v, v' \rangle), C \in L(v')$ **then** create v', add R to $L(\langle v, v' \rangle)$ and C to $L(v')$
$\mathcal{H}$-Rule	**If** $R \in L(\langle v, v' \rangle)$, $R \sqsubseteq_* S$, and $S \notin L(\langle v, v' \rangle)$, **then** add S to $L(\langle v, v' \rangle)$

$\sqsubseteq_*$ denotes the reflexive, transitive closure of $\sqsubseteq$

Subsumption

A concept D subsumes a concept C (denoted as $C \sqsubseteq D$) *iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all models $\mathcal{I}$

of $\mathcal{T}$, i.e., every instance of C must be an instance of D. Subsumption can be reduced to satisfiability, i.e., subsumes(D, C) $\Leftrightarrow \neg sat(\neg D \sqcap C)$ and $C \sqsubseteq \bot \Leftrightarrow \neg sat(C)$.

ABox

Assertional axioms include concept assertions and role assertions. A concept assertion has the form $a : C$ where $a \in N_O$ and $C \in N_C$. A role assertion has the form $(a, b) : R$ where $a, b \in N_O$ and $R \in N_R$. An ABox consists of a finite set of assertional axioms. An ABox $\mathcal{A}$ is satisfiable if an interpretation $\mathcal{I}$ satisfies all the axioms in $\mathcal{T}$ and assertions in $\mathcal{A}$, i.e., $\mathcal{I} \models a : C$ *iff* $a^{\mathcal{I}} \in C^{\mathcal{I}}$, $\mathcal{I} \models (a, b) : R$ *iff* $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $\mathcal{I} \models \mathcal{A}$ *iff* $\mathcal{I} \models \phi$ for every $\phi \in \mathcal{A}$. The interpretation $\mathcal{I}$ is called a model of $\mathcal{A}$. The ABox $\mathcal{A}$ is called consistent. The individual a is called an instance of the concept C with respect to the TBox $\mathcal{T}$ and the ABox $\mathcal{A}$ *iff* $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for all models $\mathcal{I}$ of both $\mathcal{T}$ and $\mathcal{A}$.

The satisfiability and instance problem can be reduced to the consistency problem, i.e., concept C is satisfiable w.r.t. $\mathcal{T}$ if the ABox $a : C$ is consistent w.r.t. $\mathcal{T}$, and a is an instance of C w.r.t. $\mathcal{T}$ and $\mathcal{A}$ if the ABox $\mathcal{A} \cup \{a : \neg C\}$ is inconsistent w.r.t. $\mathcal{T}$.

Knowledge Base

A main purpose of DL is to reason about a Knowledge Base (KB). A TBox ($\mathcal{T}$) and an ABox ($\mathcal{A}$) are used for describing two different kinds of statements: concepts and individuals in ontologies, both of which make up an ordered tuple $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. There exists $\mathcal{I} \models \mathcal{K}$ *iff* $\mathcal{I} \models \mathcal{T}$ and $\mathcal{I} \models \mathcal{A}$.

Here is an example of an $\mathcal{ALC}$ knowledge base, which defines both a TBox ($\mathcal{T}$) and an ABox ($\mathcal{A}$).

Example 2.1

$\mathcal{T} = \{Man \equiv \neg Woman \sqcap Person, Woman \sqsubseteq Person,$

$$Mother \equiv Woman \sqcap \exists hasChild.T \}$$

$$\mathcal{A} = \{Man(John),\ \neg Man(Monica),\ Women(Jessica),$$
$$hasChild(Monica, Jessica)\}$$

Classification

The classification of a TBox results in a subsumption hierarchy (or taxonomy) of all named concepts, with $\top$ as the root. If two named concepts A, B have a subsumption relationship, e.g., $A \sqsubseteq B$, then B is called an ancestor of A and A is a descendant of B. In case there exist no concepts A', B' such that $A \sqsubseteq B'$ and $B' \sqsubset B$ or $A \sqsubset A'$ and $A' \sqsubseteq B$, then B (A) is called a predecessor (successor) of A (B).

Additional Description Logic Constructors

$\mathcal{ALC}$ can be extended by various constructors that are denoted in the logic's name: $\mathcal{H}$ for role hierarchies, $+$ for transitive roles ($\mathcal{S}$ stands for $\mathcal{ALC}+$), $\mathcal{I}$ for inverse roles, $\mathcal{R}$ for role chain axioms ($\mathcal{R}$ includes $\mathcal{H}+$), O for nominals, Q for qualified number restrictions, $\mathcal{N}$ for number restrictions, and $(\mathcal{D})$ for using datatypes. For instance, OWL is a syntactic variant of the DL $\mathcal{SROIQ}(\mathcal{D})$ and $\mathcal{EL}$ is a subset of $\mathcal{ALC}$ supporting only $\sqcap$ and $\exists$.

Qualified Cardinality Restriction

A Qualified Cardinality Restriction (QCR) is used to specify the upper ($\leq nR.C$) or lower ($\geq nR.C$) bound on the number of R-successors of concept C, where $R \in N_R$ and $C \in N_C$. If there are two individuals x, y having $x^{\mathcal{I}}, y^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, which are related to the role R, i.e. x is a R-successor of y, *iff* $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$. The definition of all the R-successors of C for a given role R is defined as $Subs(C) = \{y^{\mathcal{I}} \in C^{\mathcal{I}} | (x^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}\}$. We use $\leq nR.C$ ($\geq nR.C$) to indicate the maximum (minimum) number of R-successors of concept C for the given role R.

2.2.2 Tableau Algorithm

A tableau algorithm decides the satisfiability of a given concept C by constructing a completion graph for C. It attempts to construct an interpretation $\mathcal{I}$ that satisfies C, i.e., there exists an instance x such that $x \in C^{\mathcal{I}}$. A complete and clash-free completion graph for C is interpreted as C being satisfiable. According to Example 2.1 mentioned above, let $\mathcal{I}$ be an interpretation with:

$Man^{\mathcal{I}} = \{John\}$

$Woman^{\mathcal{I}} = \{Jessica, Monica\}$

$Mother^{\mathcal{I}} = \{Monica\}$

$Person^{\mathcal{I}} = \{Jessica, Monica, John\}$

$hasChild^{\mathcal{I}} = \{(John, Monica), (Monica, Jessica)\}$

then it holds that $\mathcal{I} \models \mathcal{T}$ and $\mathcal{I} \models \mathcal{A}$.

A model is represented by a tableau completion graph, where concept descriptions are built using boolean operators ($\sqcup$, $\sqcap$, $\neg$), universal restriction ($\forall$), and existential ($\exists$) value restriction on concepts [6]. The tableau completion graph for $\mathcal{ALCH}$ is a labeled graph $G = \langle V, E, L \rangle$, where each node $x \in V$ is labeled with a set $L(x)$ of concepts, and each edge $(x, y) \in E$ is labeled with a set $L(x, y)$ of roles. A completion graph G contains a clash, if $\{A, \neg A\} \subseteq L(x)$ for some atomic concept A, or $\bot \in L(x)$. The completion rules for $\mathcal{ALCH}$ are shown in Table 2.2. If no completion rule can be applied to the graph G, then it is complete. Example 2.1 illustrates how the tableau algorithm determines the satisfiability of concept C defined as $C \sqsubseteq \exists R.A \sqcap \forall S.\neg A$ where R and S are roles with $R \sqsubseteq S$ and A is a concept name.

Example 2.2

First we create a, add C and its definition to $L(a)$, and apply the $\sqcap$-Rule:

$L(a) = \{C, \exists R.A, \forall S.\neg A\}$

The only applicable rule is $\exists$-Rule and we obtain

$L(\langle a,b\rangle) = \{R\}, L(b) = \{A\}$

Then we can apply the $\mathcal{H}$-Rule and obtain

$L(\langle a,b\rangle) = L(\langle a,b\rangle) \cup \{S\}$

The $\forall$-Rule is applied because $S \in L(\langle a,b\rangle), \neg A \notin L(b)$ and we obtain

$L(b) = L(b) \cup \{\neg A\}$

Finally, there is a clash because $\{A, \neg A\} \subseteq L(b)$. Therefore C is unsatisfiable because no model $\mathcal{I}$ for C can be found.

2.2.3 Transitive Closure

The transitive closure of a set X is the smallest transitive set that contains X and the transitive closure of a binary relation R on a set X is the smallest relation on X that contains R and is transitive. A relation R on a set $X = \{a,b,c\}$ is transitive, if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ and $(b^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$, then $(a^{\mathcal{I}}, c^{\mathcal{I}}) \in R^{\mathcal{I}}$.

2.3 Reasoning

Reasoning is the computation of inferences. An efficient reasoning algorithm plays an important role in inferring implicit knowledge from explicitly expressed knowledge in a knowledge base or an ontology. We mainly focus on tableau-based reasoning, which is usually used in DLs. As a comparison with tableau-based reasoning, two other types of reasoning are introduced: RDF reasoning and resolution-based reasoning.

2.3.1 RDF Reasoning

The Resource Description Framework (RDF) with its vocabulary description language RDF-Schema (RDFS) constitutes the basic language for semantic web. RDF is a graph-based data model, which is used for metadata of web resources and generated structured information. We usually use Uniform Resource Identifier (URI) to create globally unique concept names for resources. RDFS is RDF with a special vocabulary for terminological knowledge, which can be used to express terminological knowledge of classes and properties in hierarchies. However, it is not possible to define negation of expressions, cardinalities, a set of classes, metadata of the schema with RDF reasoning. Therefore, sometimes we need to provide a specialized inference engine to support reasoning in RDFS, such as Jena, a semantic web framework for Java [11].

2.3.2 Resolution-based Reasoning

Resolution-based methods with DL ontologies, which is applied to general first-order theorem proving always have the worst-case optimal complexity. By using resolution theorem provers and redundancy elimination rules, the resolution-based methods can be implemented efficiently and reduce the search space of reasoners. It has been implemented in many practical systems such as Vampire [56], which is a resolution-based theorem prover for first-order classical logic. However, compared with tableau-based reasoning, which needs blocking techniques to ensure termination of the system, there is no guarantee for termination since first-order logic is semi-decidable, which may lead to non-termination [40].

2.3.3 Tableau-based Reasoning

Tableau-based methods for satisfiability checking are widely used as one of the major techniques for ontology reasoning systems. Through constructing a model of input formulas and checking whether it contains a contradiction, the satisfiability of the model can be concluded. Tableau-based reasoning can be efficiently implemented by using appropriate optimization techniques. In practical systems, tableau-based methods have been successfully implemented in some ontology reasoning systems such as FaCT++ [55], which is a DL reasoner platform designed for experimenting with tableau-based methods [55]. Moreover, tableau-based reasoning can be used to solve many other problems such as the consistency problem for ABoxes and TBoxes, subsumption and instance problems [39].

2.4 Reasoning Systems

One of the reasoning components in DL systems is an engine known as classifier which infers entailed subsumption relations from knowledge bases. Research for most DL reasoners is focused on optimizing classification using one single processing core [7, 24, 16]. Considering the ubiquitous availability of multi-processor and multi-core processing units not many OWL reasoners can perform inference services concurrently or in parallel.

2.4.1 Sequential Systems

There are some existing description logic reasoners which apply sequential computation methods. For example, the OWL reasoner HermiT [22] is based on a novel hypertableau calculus to classify a number of ontologies in a sequential way; Racer [25] is a

knowledge representation system that implements a highly optimized tableau calculus for the description logic $\mathcal{SRIQ}(\mathcal{D})$; FaCT++ [55] implements a tableau decision procedure for the well known $\mathcal{SHOIQ}$ description logic, with additional support for datatypes, including strings and integers in its sequential computation system. Because of different optimization techniques applied on the DL reasoners, the subsumption hierarchy of an ontology can be computed and constructed efficiently by using various methods.

2.4.2 Concurrent Systems

Concurrency is a popular property of current systems. It allows one to execute several computations simultaneously and potentially interact with one another. During the execution, the computations may share the same processor with time-shared threads, which can practically reduce the time complexity such as in the ELK reasoner [30], which has successfully implemented a concurrent classification algorithm for DL $\mathcal{EL}_+$ ontologies. Concurrent execution of shared resources and interactions may lead to deadlocks or starvation especially when the system is extremely large and complex. Therefore, the design of concurrent systems needs to use extra processing techniques for coordinating execution and data exchange to achieve a better performance.

2.4.3 Distributed Systems

A distributed system is a group of networked and possibly heterogenous computers with independent processors and memory used to distribute the whole task to several processors to be executed simultaneously. During the execution, all processors can update and exchange information. For example, the novel reasoning system Konclude [53] implements different reasoning procedures and optimizations techniques.

The common approach of *semantic mapping* is used to define the semantic relations between concepts belonging to different ontologies. However, it cannot guarantee the capability of reasoning within a system containing multiple interconnected ontologies. Therefore, a reasoning approach, which encodes both ontologies and mappings into a unique architecture has been proposed, i.e., Distributed Reasoning Architecture for a Galaxy of Ontologies (DRAGO), which implements such distributed decision procedure [52].

2.5 Parallel Computing

2.5.1 High-performance Computing

High-performance Computing (HPC) is used for solving complex computing problems by applying parallel techniques on super large-scale computers. Due to the better performance of HPC systems, they mainly focuses on solving the problems of supercomputing, simulation and analysis to maintain and balance the constant resources among different processors concurrently.

HPC systems employ various different parallel processing applications and techniques to achieve better performance, such as hardware virtualization [37] to enhance the productivity of HPC applications; hyper-threading [33] to support operating systems with scheduling tasks and affect the overall performance; virtual machine [28] to secure, manage and migrate the HPC applications, and power-aware run-time system [27] to achieve better impacts of power reduction and energy savings.

2.5.2 Atomic Data

Different approaches have been proposed to create non-blocking algorithms and ensure data integrity in concurrent environments. However, most algorithms solve the problems of memory location, concurrent access and information exchange among multiple threads using locks, which can result in lost updates or a deadlocks. [13]

Atomic from the Java Concurrent package can support generating more than one thread to maximize CPU utilization and manage concurrency without locking. In a concurrent system, processes can access a shared data structure at the same time. In order to ensure data consistency and avoid conflicts among multiple processes, atomic operations is introduced as an algorithm, which is not only lock-free requiring partial threads for constant progress but also wait-free for updating information [23]. Therefore, using an atomic shared-memory structure makes sure that the concurrent approach is a non-blocking algorithm, which can process and schedule threads simultaneously.

2.5.3 Work-Stealing

In parallel computing, work-stealing is a scheduling strategy to solve the problems of dynamic multiprocessing computation. Each processor has a queue containing a list of tasks to be performed. During the processing, all the tasks in different queues will execute in parallel according to the given order. In addition, a task can also spawn new subtasks which execute in parallel with other tasks on the list. When a processor has finished all the tasks in its queue, the current available processor can steal tasks assigned to other processors. Therefore, the scheduling strategy can make sufficient use of the available processors and improve the execution time and load balancing of parallel computing.

In recent years, many different scheduling techniques were proposed and applied to achieve a better performance of multithreaded computations on parallel computers. For example, a scalable work stealing approach [18] focuses on the scalability of work stealing on distributed memory systems with high efficiency and low overhead performance; data locality of work stealing [1] presents the scheduling strategies applied on hardware-controlled shared-memory machine and the improvements of locality-guided performance; Dynamic circular work-stealing deque [14] is a lock-free algorithm, that stores the elements in a cyclic array when it overflow and requires memory which is linear to the number of deque elements.

2.5.4 Hyper-Threading

Hyper-threading is used for parallelization of computation by simultaneous multithreading. In most circumstances, there are two virtual cores which share the workload among all the processors when possible. Using hyper-threading can increase the number of independent instructions and execute separate data and instruction streams in parallel, which not only increases the flexibility of scheduling and also lowers the influence of data latency by using internal resources to achieve a better performance. Hyper-threading has various functions, which allows concurrent scheduling of two or more processes per core and shares the same resources with other processors. In addition, if the resource is not available for the current process, it still can be available during the procedure of other processes to share the resources dynamically [34].

Hyper-threading mainly has three different ways to manage resources. Firstly, using partitioning to allocate half of the resources to each logical processor, which can avoid latency and improve the utilization of different structures. Secondly, there is a threshold when hyper-threading is applied. The threshold has the advantages of sharing flexible resources with limited resource usage. Therefore, for some structures

with less available resources, using hyper-threading can lower the occupancy time and balance occasional high utilization of different processes. However, due to applying round robin techniques, there could be a threshold which prevents one logical processor from all the other available resources. Thirdly, full sharing is ideal for flexible resources sharing without limitations, especially for large structures. [35, 32]. In recent years, new methods have been proposed to improve the performance of hyper-threading, such as process scheduling heuristics [12] and multi-level threading [15]. Therefore, hyper-threading technology can improve the performance of parallelization even with limited resources and available processors in various circumstances.

3 Background and Related Work

3.1 Introduction

In this chapter, different classification methods and current popular DL reasoners are presented. First, we focus on the development of sequential methods and the improvements of them. Second, some parallel classification methods from relevant research are introduced with their evaluation. Finally, we review three complete DL reasoners implemented with different parallel techniques.

3.2 Sequential Classification Methods

The construction of concept subsumption hierarchies is important to find an efficient method to reduce the cost of classification and time of computation. In this part, we focus on different classification methods for the sequential computation of concept subsumption hierarchies [5, 51, 21]. First we introduce the Brute Force method, top search and bottom search. Based on the basic search methods, improved methods are described. The simple traversal method and enhanced traversal method will be introduced as improvements in the search process.

3.2.1 Brute Force

Brute Force always comes first as the basic classification method. In this method, we use the original way to compare each concept with the one to be inserted. After having performed all the possible comparisons, each concept has its immediate successors and predecessors, the subsumption hierarchy is constructed according to the subsumption relationships among the concepts.

In the top search, a given concept c is compared with all other concepts. If a subsumption test succeeds, the predecessor concept is added to the set $Pre(c)$, which contains all the predecessors of c. The bottom search works in a dual way. Through subsumption tests with all other concepts, all the successors of concept c are added to the set $Succ(c)$.

According to the sets of successors and predecessors for each concepts, the method checks all the concepts to find immediate predecessors and successors for the concepts. Finally we can construct the subsumption hierarchy and put c in the exact position. The time complexity of this method is $\Theta(n^2)$ for both the worst and the best case. Moreover, one needs to perform subsumption tests twice for each concept to find its immediate predecessor and successor concepts in the hierarchy. Therefore, it is advisable to compare each concept along the hierarchy to avoid unnecessary comparisons and reduce the number of subsumption tests.

Based on the Brute Force method, search methods were adopted to check all the concepts in the hierarchy and find the exact position for the given concept. During the search process, if the current concept is not a successor (predecessor) of concept c, then we can conclude that all the successors (predecessors) of the current concept are not related to concept c. Following this method, we discuss about about the basic top search and bottom search method in the following.

For a given concept c with concept $x \in X$, where X is the set of all concepts in

the subsumption hierarchy, we use top search and bottom search to construct the subsumption hierarchies. [5]

Top Search Top search begins from the top concept ($\top$). Following all the successors, the given concept c is compared with each concept along the current hierarchy until all the predecessors x of c have been found and added to the set $Pre(c)$.

Bottom Search In the bottom search, the given concept c moves up from the bottom concept ($\bot$) by following all the predecessors and is compared with each concept along the current hierarchy until all the successors x of c have been found. Then all the successors are added to the set $Succ(c)$.

As a result, both top search and bottom search methods can reduce the number of comparisons and subsumption tests. However the time complexity is still $\mathcal{O}(n^2)$ for the worst case. If we could record that a concept has been visited and has a relationship with c, then we can reuse the information of concepts which have been executed before. The efficiency of the search method can be improved.

3.2.2 Simple Traversal Method

In order to reduce the number of subsumption tests and the average time complexity, the simple traversal method creates three labels 'Visited', 'Negative' and 'Positive' which can been used to denote the information of each concept. The label 'Visited' is used to record that a concept has been visited. 'Positive' and 'Negative' are used to indicate the subsumption relationship between c and the concept which has been tested. Using these labels, the improved top search and bottom search methods are introduced as follows. [5]

Top Search Before a comparison, the top search method first marks the current compared concept with the label 'visited'. Then it begins with the traversal method from the top. The procedure assumes that c is subsumed by x. For each successor $y \in Succ(x)$, the simple-top-subs method is called to find the relationship between c and a successor y of x. If it has done the subsumption test before then the simple-top-subs method had marked y with the label 'negative' or 'positive'. If it is marked 'positive', y is added to the set *Pos-Succ*. If y has not been marked, then the *subs*?() method is called to test the relationship between y and c and return the result. The procedures are shown in Algorithm 1, 2 which are adapted from [5].

An example is shown in Figure 3.1. The top-search method begins with $x = \top$. For the direct successors X_1 and X_2 of $\top$, call the simple-top-subs method to check whether they subsumes c. Since each concept has its label, if X_1 has been visited before, then we need to check its label. Otherwise, call the subs? method to test the relationship between X_1 and c. If we find X_1 and X_2 are 'positive', then test all its successors. After the tests, if both Y_1 and Y_2 are marked 'negative', but X_1 is 'positive', X_1 will be added to the *Pos-Succ*. For the direct successors Y_3, Y_4 of X_2, the test results show that Y_3 and X_2 are 'positive' in contrast to Y_4. In conclusion, c is inserted as the successor of X_1 and Y_3 shown in Figure 3.1.

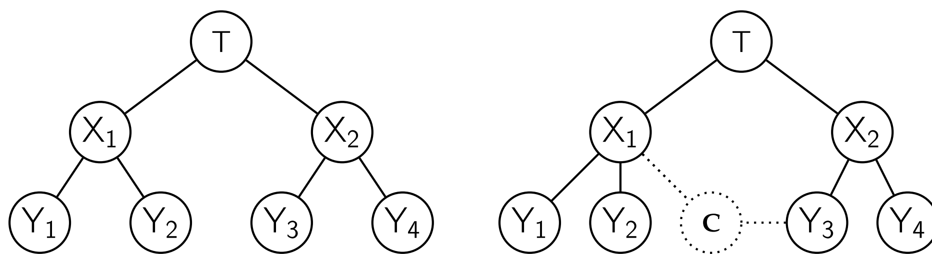


FIGURE 3.1: Insert concept c through top search of simple traversal method

Algorithm 1: top-search(c,x)

```
1  Input: c - concept to be inserted; x - current compared concept
2  Output: Pos-Succ - list of all the positive concepts compare with c.
3  mark(x, 'visited');
4  for all y ∈ Succ(x) do
5      if  simple-top-subs?(y, c) then
6          Pos-Succ ← Pos-Succ ∪ {y};
7      if Pos-Succ is empty then
8          Result ← {x};
9      else
10         for all y ∈ Pos-Succ do
11             if not marked?(y, 'visited')
12                 Result ← Result ∪ top-search(c,y);
```

Bottom Search Bottom search performs in a dual way. The search begins from the bottom of the hierarchy with $x = \bot$. For each concept x of X, it checks whether the current element c is subsumed by its predecessor $y \in Pre(x)$. It works symmetrically to the top search method.

Due to the use of labels in subsumption hierarchies, this method can usually reduce the number of subsumption tests by marking each concept with labels to avoid visiting the same concept again. Compared to the brute force method, it reduces the comparison time for the same concept in the hierarchy. But it still has the time complexity $\mathcal{O}(n^2)$ for the worst case. If each step can reuse the information on tests that been performed before and the bottom search also uses the results from the top search, then it will be more efficient.

Algorithm 2: simple-top-subs(y, c)

1 Input: c - concept to be inserted; y - current compared successor of x.
2 **if** marked?(y, 'positive') **then**
3 $\quad Result \leftarrow true$
4 **else if** marked?(y, 'negative') **then**
5 $\quad Result \leftarrow false$
6 **else if** test if y subsumes c **then**
7 $\quad$ mark(y, 'positive')
8 $\quad Result \leftarrow true$
9 **else**
10 $\quad$ mark(y, 'negative')
11 $\quad Result \leftarrow false$

3.2.3 Enhanced Traversal Method

Top Search To use negative information during the top search, we need to check all predecessors z of y if a test sub?(z, c) has failed. Then it is not necessary to perform the expensive subsumption test to conclude $y \notin Pre(c)$. The enhanced-top-subs method makes sure that the subsumption tests for all predecessors of y have been performed before y.

To use positive information during the top search, we need to find out whether there is a successor z of y and $z \in Pre(c)$ before the test of checking $y \in Pre(c)$. If z exists, we can conclude that $y \in Pre(c)$ without a subsumption test. If the call sub?(y, c) returns true, then y and all its predecessors are marked 'positive'. The enhanced-top-subs method tests all the predecessors before making a subsumption test. It is more efficient to propagate positive information up through the subsumption hierarchy resulting in a smaller number of subsumption tests. The enhanced-top-subs phase is shown in Algorithm 3 which is adapted from [5].

Algorithm 3: enhanced-top-subs(y, c)

```
1  Input: c - concept to be inserted; y - current compared successor of x.
2  Output: If return true, then y is marked 'positive'; otherwise return false.
3  if marked?(y, 'positive') then
4      Result ← true
5  else if marked?(y, 'negative') then
6      Result ← false
7  else if for all z ∈ Pre(y)
8          enhanced-top-subs?(z, c) &
9          subs?(y, c)
10     then
11         mark(y, 'positive')
12         Result ← true
13 else
14     mark(y, 'negative')
15     Result ← false
```

The first example is shown in Figure 3.2. The top search using negative information tests X_1, but before testing Y, its direct predecessors X_1, X_2, $X_3 ... X_m ... X_n$ are tested. As a result, if there exists a predecessor of Y that is negative, then Y is negative. The top search using positive information, first tests X_1 and then Y. The positive result of Y can be propagated to X_1, X_2, $X_3 ... X_m ... X_n$. The second example shown in Figure 3.3 uses negative information, it first tests X_1 and before testing Y_1 its direct predecessor X_2 is tested. If both X_1 and X_2 are negative, then we can conclude that all their successors are negative. On the contrary, when using the positive information, first test X_1 then all its successors Y_1, Y_2, Y_3 and Y_4, and finally X_2. If all the successors are positive, the positive information can be propagated up to X_2 which is also positive.

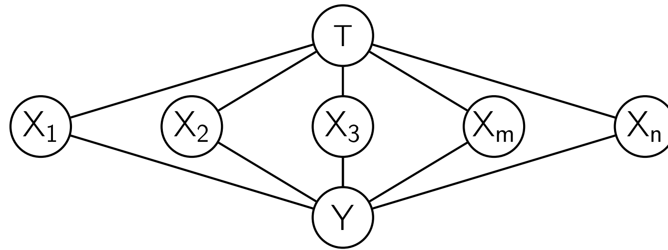


FIGURE 3.2: Inserted concept c is a direct successor of Y

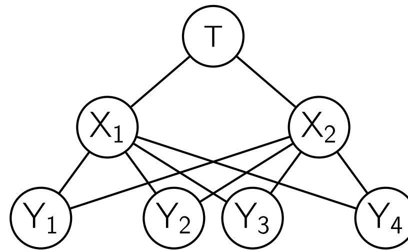


FIGURE 3.3: Concept c is not a direct successor in the hierarchy

Bottom Search Bottom search can use the information from tests which not only have been performed previously, but also resulted from the top search as well. From the top search, if we use the positive information and find the successor z of y and $z \in Pre(c)$, then we can prove that z is a predecessor of c without a subsumption test.

Due to reusing all the accumulated information, this method is more efficient than the simple traversal method. When compared with the simple traversal method, the number of necessary comparison operations can be reduced. This method does not require to test all the predecessors before testing a concept by propagating positive information up and negative information down the hierarchy. If one concept is marked 'negative', then all the successors can be marked at the same time. In this case, if we could figure out the possible subsumption relationships from the existing information, then the number of subsumption test can be reduced by transitive closure. [5, 51]

3.2.4 Optimized Classification Method

Based on the enhanced traversal method, the optimised classification method applies the transitive closure reduction and constructs the subsumption hierarchy in an indirect way by creating two sets: the known (K) and remaining possible (P) subsumer pairs. It performs subsumption tests to increase the set K and reduce the number of pairs in the set P until P becomes empty and K includes all relations in the subsumption hierarchy. The method reuses the information from previous subsumption tests according to the enhanced traversal (ET) method and exploits the transitivity of subsumer from P to K without actual reasoning. [21]

For example: if we know that $\{\langle C, D\rangle, \langle E, F\rangle\} \subseteq K$ and try to add $< D, E >$ to K, then we can add $< C, F >$ to K according to the transitivity relationship. On the contrary, if $\{\langle C, D\rangle, \langle E, F\rangle\} \subseteq K$ and $\{\langle D, E\rangle, \langle C, F\rangle\} \subseteq P$, and $\langle C, F\rangle$ needs to be removed from P, then $\langle D, E\rangle$ should be removed from P at the same time which is shown in Figure 3.4. Adding and removing sets from P, which contains possible subsumption pairs, can exploit the subsumption relationships with a reduced number of subsumption tests to improve efficiency.

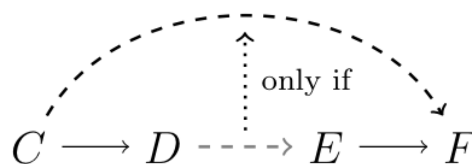


FIGURE 3.4: Possible Relationships: nodes represent classes, solid edges represent pairs in K, and the light dashed line represents a pair that can be in P only if the pair represented by the dashed line is in P or K. (Adapted from [21])

The classification phase using the modified version of ET method decides all the possible pairs in P which actually hold. For each class C, it constructs a pre-model which satisfies C. $P|_c$ denotes the possible subsumer set of C. If there is no class

$D_{\mathrm{i}} \in P|_{\mathrm{c}}$, then remove $\langle C, D_{\mathrm{i}} \rangle$ from P. For all the existing classes $D_{\mathrm{i}} \in P|_{\mathrm{c}}$, we build a subsumption hierarchy of C (H_{c}) and a queue Q which contains all the successors in $\mathrm{H_c}$ after traversal. When we remove the head of Q, the method applies the transitivity relationship we mentioned in Figure 3.4. If there exists a pre-model which satisfies C but not D, then all the subsumers in D but not C can be removed from H_{c} and $P|_{\mathrm{c}}$. Otherwise, if the subsumption between C and D holds, this is recorded in K and each successor E of D in H_{c} can be added to Q. In this case, if we find $\langle E, F \rangle$ and $\langle C, F \rangle$ exist in K and $\langle D, E \rangle$ in H_{c}, then the conclusion should be $\langle D, E \rangle$ can be added to the set K. It performs iteratively until P becomes empty and K includes all the potential pairs according to this optimised classification method.

This method includes only the top-down and not the bottom-up phase. Therefore, during the classification it only needs to find the successors for each class. It can be more efficient when compared with ET method. But the process of deciding which possible pair in P belongs to K is time consuming. It is necessary to check each pair with all the compared classes, which takes time to find the possible subsumers. It is more efficient when there are more tests for possible subsumptions with less compared concepts.

In this chapter, we have introduced the sequential classification methods in semantic web. Each method focuses on the reduction of subsumption tests and time complexity. In the next chapter, based on the sequential methods parallel classification methods will be presented.

3.3 Parallel Classification and Reasoning Methods

Parallelization is a useful technique to speed up the performance of classification by working simultaneously. Due to the increasing scale of ontologies and their number of classes, it is promising to use some parallel methods to speed up reasoners. For parallel classification, there are two important parameters: partition size and number of threads for parallelization. Based on the sequential classification method, in the past various parallel reasoning methods have been proposed: A distributed reasoning architecture to accomplish reasoning through a combination of multiple ontologies interconnected by semantic mappings [52]; A research methodology for scalable reasoning using multiple computational resources [57]; A parallel TBox classification approach to build subsumption hierarchies [3]; An optimized consequence-based procedure using multiple cores/processors for classification of ontologies expressed in a tractable fragment of OWL [29]; [36] applied some computation rules in a simple parallel reasoning system; A parallel DL reasoner for $\mathcal{ALC}$ [60, 61]; Merge-based parallel OWL classification [62]; A rule-based distributed reasoning framework that can support any given rule set [42]; a framework to formalize the decision problems on parallel correctness and transfer of parallel correctness, providing semantical characterizations, and obtaining tight complexity bounds [2].

In this part we first review some parallel methods: parallel classification [3], merge classification [61], a scalable and parallel reasoning approach [57] and a distributed reasoning architecture [42], then we evaluate their efficiency.

3.3.1 Parallel TBox Classification Algorithm

The parallel TBox classification method [3] can be divided into three generations. The first generation is a set of sound and incomplete algorithms. To improve the completeness, two scenarios that cause incompleteness will be discussed in the second generation which achieves the sound and complete parallel classifier algorithm. The third generation implements concurrent TBox Classifier to classify the TBox concurrently and achieve the sound and complete algorithm that can efficiently process much bigger ontologies.

First Generation In the first generation, the parallel classification method uses a shared-memory approach and one global tree to manage the concurrency. The initial taxonomy is created from a list containing already known predecessors and successors that is sorted in topological order. From the list the classifier assigns random partitions of concepts for each thread to execute the partition of the whole task. The threads take turns and work concurrently until the whole task has been finished. When inserting or updating information among the threads and constructing one global subsumption tree, it uses the lock mechanism to ensure data integrity for the changes of hierarchy information during the construction. In order to avoid unnecessary traversals and tableau subsumption tests, the classification adopted ET method when computing the subsumption hierarchy.

In this generation, the incompleteness is caused by classifying random partitions of concepts for each thread. Due to the variety of ontologies, it is possible that some subsumptions are missing during the procedure. Therefore, it is a sound but incomplete algorithm in the first generation.

Second Generation In order to demonstrate the soundness and completeness of this method, there is a small example with 16 concepts. For each thread, it includes a partition of concepts according to the topological-order list. The division for each thread is shown as follows and the concepts in brackets stand for synonyms.

thread#1 (female not-male), girl, parent

thread#2 woman, mother, (male not-female)

thread#3 man, boy, father

thread#4 not-boy, not-father, not-girl

thread#1 not-man, not mother, not-parent, not-woman

The parallel classification procedure performs in a round-robin manner. The threads are activated with their assigned partition and work in parallel. This method focuses on the subsumption tests for each thread and merges all of them into one global hierarchy. It is necessary to consider the missing subsumptions among the threads. Like the example above, there is a concept (female not-male) in *thread*#1 which has a subsumption relationship with concept (woman) in *thread*#2. Therefore, we need to notify *thread*#2 when the concept (female, not-male) has been inserted by *thread*#1. Moreover, for each newly inserted concept, we should also consider subsumption relationships with the existing concepts in other threads.

In this generation, there is a global array for every newly inserted concept, which is indexed by thread identifications. Therefore, the concepts in the global tree as well as newly added concept (parent) in the array are locked for modification. For each newly inserted concept, the classifier first performs the top-search phase, which sets the parents (not-girl), (not-boy) and adds them to the list of children for each parent. Then it calls the bottom-search phase, which sets the children (woman), (man) and updates the parents' children correspondingly. After searching, it needs to check again whether other threads updated their index and repeat the search methods to get the

final global array. As a result, other threads are notified of the newly added concept (parent). It works iteratively for each newly inserted concept. The final complete subsumption hierarchy of the example is shown in Figure 3.5.

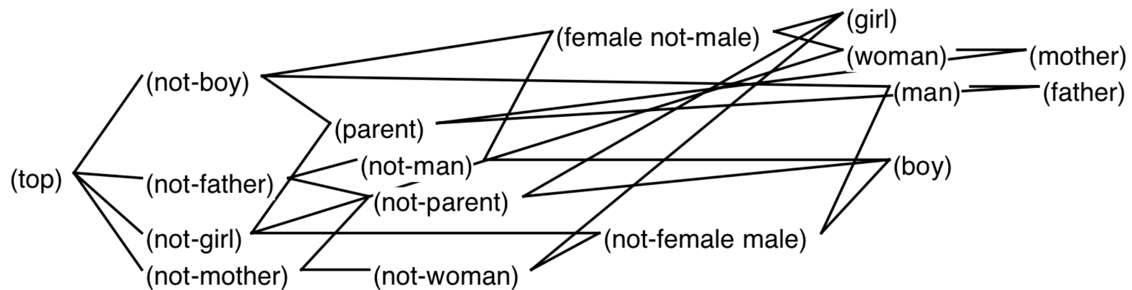


FIGURE 3.5: Final complete subsumption hierarchy (adapted from [3])

The second generation is a sound and complete algorithm for the parallel TBox classifier when concepts are inserted in parallel by different threads. It uses a lock mechanism for updating the information of a concept in the global subsumption hierarchy.

Third Generation The third generation of the algorithm uses concurrent TBox Classifier, which is more effective compared with previous generations. To improve the partitioning phase of this method, it implements the informed partitioning algorithm, which can reduce the number of repeated executions of the search methods. In order to avoid inserting a concept redundantly into different informed partitions, all the concepts which have known interactions will be placed in the same partition. There is a ignore-list, which contains all the concepts in the same partition to avoid adding a concept redundantly into different partitions. Each time before the search method is called for a new concept from the list, it will be checked if it has already been placed in the ignore-list.

The classifier works in a round-robin manner to assign partitions for each idle thread. It also shares a global array, which is used for notifying other threads when a

new concept has been inserted. During the construction of the global taxonomy, it uses lock-free data structures among all the partitions. If a new concept (female, not-male) has not been inserted but has interactions with a concept (not-boy) in the array, it will be added to the partition of *thread*#4 and deleted from other partitions, i.e., *thread*#1, at the same time. Otherwise if it exists in the array, the concept can be inserted into the taxonomy directly. In addition, if the inserted concept (not-girl) has many interactions with concepts in the hierarchy such as (parent), (woman) and (not-female, male) and might need more reruns before finding its exact place in the hierarchy, we postpone it and add it to the waiting list, which contains the concepts to be inserted later. Therefore it is a sound and complete algorithm for the concurrent TBox classifier.

This method uses three generations to achieve a sound and complete algorithm for both parallel and concurrent TBox Classifiers. It improves the efficiency of the classification procedure when compared with the sequential cases. However other factors such as different partition sizes, number of threads and overheads can impact the efficiency of the parallel methods. Moreover, it is necessary for us to consider some specific optimization techniques for parallel classification to avoid subsumption tests and reduce the repeated number of search methods.

Evaluation

The parallel TBox Classifier has been used to speed up the classification process especially for some large ontologies. Using the parallel threads with a shared memory, the evaluation results are focused on the number of performed subsumption tests with a collection of 8 available large ontologies. Given the performance result from [3], it shows that using two threads the maximum of number of subsumption tests for all ontologies can be reduced to almost one half when compared to the sequential case with a small overhead. Furthermore, if partitions have interactions with other threads

as little as possible, then the overhead and the number of subsumption tests can be reduced significantly. As a result, the parallel TBox Classifier is a promising techniques for parallelization and can be improved with different configurations of threads and partition sizes on large ontologies.

3.3.2 Merge Classification

The merge classification method [61] got the ideas from the known merge sorting algorithm. It implements a heuristic partitioning scheme and divide-and-conquer (D & C) algorithm, which is based on multi-thread recursion, works recursively by dividing an original problem into two or more sub-problems and solving each sub-problem independently. The solution of the original problem needs to combine all of the results of sub-problems together. Based on D & C algorithm, this method divides an ontology into sub-domains and constructs the subsumption hierarchy using different threads. After all partial hierarchies have been constructed, the final hierarchy is computed by one processor by merging all the partial hierarchies together. The process can be divided into two phases: divide and conquer phase and combining phase.

Divide and Conquer Phase In the divide and conquer phase, the domain Δ first is divided into smaller partitions of sub-domains Δ_i for each thread. Using classification computations, each sub-domain is executed in parallel. In this method, the divide operation implemented heuristic partitioning techniques for partitioning over Δ. The conquering operation uses the classification methods, top search and bottom search to determine the immediate predecessor and successor of each concept in Δ_i and construct the classified concept hierarchy for each thread.

Combining Phase In the combining phase, the classified sub-taxonomies will be merged together. Using a modified top merge method to merge two sub-domains: Δ_α and Δ_β, the method calculates each concept in Δ_β to find the immediate predecessor in α. Then using the improved bottom merge method it finds the immediate successor of the concept in Δ_β into Δ_α. After combining each sub-domain into one domain, all the sub-domains can be merged into the final subsumption hierarchy in the end. Here is an example to illustrate the algorithm further.

Example 3.1: We use the TBox shown in Figure 3.6. It contains simple concept subsumption axioms and entails the subsumption hierarchy shown in Figure 3.6 on the right. In the divide phase of the algorithm, the concepts are divided into two subgroups: G_1 = {A_2, A_3, A_5, A_7} and G_2 ={A_1, A_4, A_6, A_8} as shown in Figure 3.7.

$\mathcal{T}$:

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

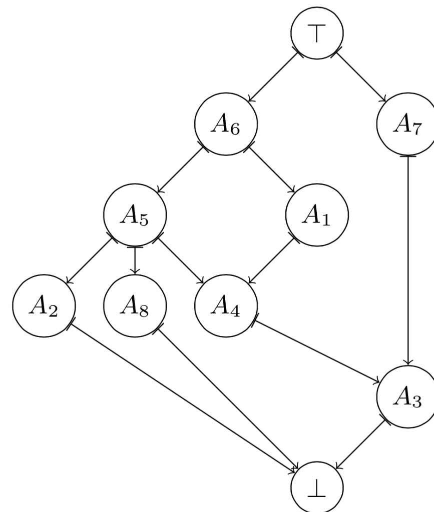


FIGURE 3.6: The given TBox and the classified terminology hierarchy

After each division group has constructed its own subsumption hierarchy (see Figure 3.7), the merge phase will merge the two groups into one hierarchy. For instance, in order to insert the concept A_4 into the α subsumption hierarchy, the top merge method is called to compute the immediate predecessors of A_4 and finds no such predecessor in Δ_α. Then the bottom merge method is called and determines that A_5 in Δ_α is the

immediate predecessor of A_4. Finally a possible computation path from A_5 to A_4 is determined. In the example, we find that A_4 is subsumed by A_5 and add A_4 to the subsumption hierarchy. After merging all concepts into one hierarchy, the result is shown in Figure 3.7.

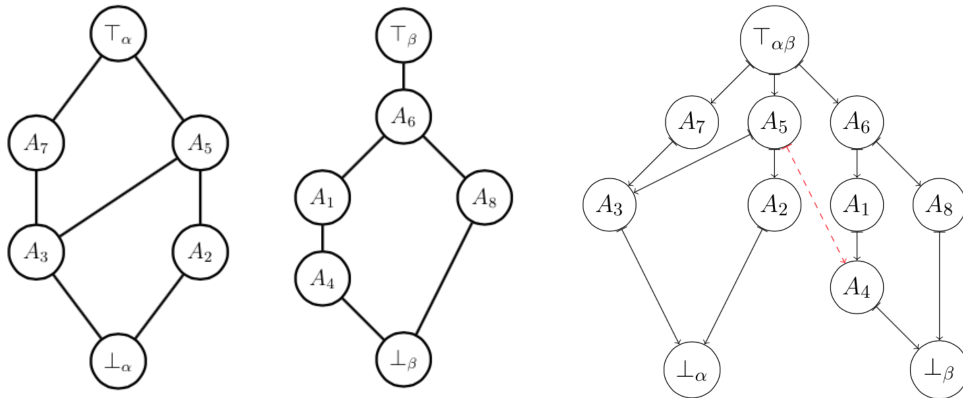


FIGURE 3.7: The subsumption hierarchy divisions of different groups (left two) and the subsumption hierarchy after merging both (right)

This method implements the algorithm by dividing all the concepts into different groups and merging the independent subsumption hierarchies together. The divide and conquer phase divides all the concepts into different groups and executes the subsumption tests to construct the subsumption hierarchy for each thread. The merge phase merges all the hierarchies into one processor and checks the relationships between concepts in the different groups to build a complete subsumption hierarchy. In both phases it is necessary to test the relationships between each concept, which affect the efficiency of this method. Therefore, it is important to find the appropriate partitions of concepts for each group which can reduce the overhead and the number of subsumption tests in the process.

Evaluation

The merge classification method uses a multi-threading model which is supported by multi-processor computing facilities. Given the experimental results from [62], it was

conducted on a 16-core computer running Solaries OS and Sun Java. Although the big complex ontologies need longer single thread computing time, the observed scalability is linear and the overhead can be reduced significantly. The results show that it has a better scalability especially on more difficult and bigger ontologies. Moreover, with an increasing number of the threads the reasoning performance can also remain stable. We could expect a further scalability improvement could be achieved by using more processors with advanced multi-processor computing facilities.

3.3.3 Scalable and Parallel Reasoning Approach

In this approach, a distributed method is proposed where a very large amount of tasks are distributed and executed simultaneously on independent machines. For the reasoning part, it is mainly based on monotonic rule-based reasoning. Since distributed approaches require more exchange of updated data and increase the overhead compared to using a single machine, this method was implemented with the Ibis [43] framework to deal with the problems of communication between the nodes and the heterogeneity of the systems.

This method includes three phases. In the first two phases, according to the study of existing parallel programming models, a reasoning algorithm which matches the chosen programming model is selected and uses different logics. In the last phase, a series of experimental results are performed on the cluster. The performance is evaluated by extending the complexity of logics to find out the most suitable programming model for each parallel reasoning. The results shows that using the MapReduce [41] programming model achieved linear scalability, however, the method cannot be extended to complex logics for reasoning due to the optimizations applied in this approach require some specific characteristics of RDFS [59, 57].

3.3.4 Distributed Reasoning Architecture

This architecture is mainly to solve the problems of reasoning with multiple ontologies interconnected by semantic mappings. In comparison to the global reasoning approaches, this method applied distributed reasoning techniques, which execute each ontology using a independent processor to speedup the whole process. According to the definition of distributed description logics in [10], which provides a syntactical and semantical framework for ontologies using DL theories linked by semantic mappings using collections of rules, this approach defined a distributed tableau-based reasoning procedure extended to standard DL tableau reasoning.

Architecture

This reasoning system uses a peer-to-peer network to distribute ontologies to different reasoning peers, which can register a stand alone ontology as well as an ontology with a set of semantic mappings. Each reasoning peer has two different services : registration and reasoning services.

- Registration Service is used to record and update the registered ontologies and their assigned mappings. It is controlled by the registration manager.
- Reasoning Service includes checking concept satisfiability, ontology consistency and entailment, construction of taxonomy.

These two services are connected by a registration manager, which can check the availability of memory, assign a different mapping to each ontology, and analyze parsing data by using an available distributed reasoner [52].

Compared to other DL reasoners, such as Racer [25], Fact++ [55], this approach can accommodate more reasoning capability with multiple ontologies linked with semantic mappings by using distributed reasoning techniques.

3.4 Parallel Reasoning Techniques

3.4.1 MapReduce

MapReduce is a programming model and software framework for distributed processing and generating large data sets on clusters of machines. Using MapReduce the system can automatically parallelize computations across large-scale clusters of machines and schedule message exchanges to make efficient use of the network and disks. The MapReduce framework has been successfully applied for computing RDF Schema closure and for reasoning with OWL Horst [58].

Programming Model

MapReduce is a programming model for distributed processing of data on a cluster of machines (each machine called a node) [17]. The data is divided into several partitions, and each partition is assigned to an idle node. There are three types of nodes with their own function.

Master: The master node assigns partitions to map nodes and passes the intermediate output locations to reduce nodes.

Map: The map nodes receive the partition from the Master and generate intermediate output according to the map fuction, which is used to generate and return a set of intermediate key/value pairs by processing a key/value pair. The output pairs are stored on local disks and the location of the data is returned to the Master.

Map: $(k_1, v_1) \rightarrow list(k_2, v_2)$

Reduce: The reduce nodes are notified of the locations of intermediate output. They group the values by key and process the values according to the reduce function, which is used to merge the output associated with the same intermediate key into a smaller set of values.

Reduce: $(k_2, list(v_2)) \rightarrow list(v_3)$

MapReduce for $\mathcal{EL}_+$

In this part, we will give an example of converting the completion rules into MapReduce algorithms to compute the closure of an ontology $\mathcal{O}$. S, R, and P are sets (or maps for MapReduce), where $S(X)$ maps a class name X to a set of class names, $R(r)$ maps each role name r to a set of class name pairs and P is an extension of the function R. A, B, C and D are concepts. Initial settings are $S(A) = \{A, \top\}$ and $P(A) = \emptyset$, for each class name A including $\top$ and $R(r) = \emptyset$ for each role name r. All the expressions of the form $A \in S(X)$, $(A, B) \in P(X)$ and $(X, Y) \in R(r)$ are considered as axioms. The completion rules R1-1 and R1-2 and the keys are defined in Table 3.1 [41].

Name	Normal Form	Completion Rule	Key
$R1-1$	$A_1 \sqcap A_2 \sqsubseteq B$	**if** $A_1 \in S(X)$ and $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{O}$, **then** $P(X) := P(X) \cup \{(A_2, B)\}$	A_1
$R1-2$	$(A, B) \in P(X)$	**if** $A \in S(X)$ and $((A, B) \in P(X)$ or $A \sqsubseteq B \in \mathcal{O})$, **then** $S(X) := S(X) \cup \{B\}$	A

TABLE 3.1: The completion rules for $\mathcal{EL}_+$ and keys for applying MapReduce

The behaviour of R1-1 and R1-2 can be illustrated using the axioms: $A \sqcap B \sqsubseteq C$, $A \sqsubseteq B$ and $A \sqsubseteq D$. We can infer that A is a subclass of both C and D, which can be obtained by using R1-1 and R1-2 alone. When the algorithm is initialized, $S(X)$ = $\{X, \top\}$ and $P(X)$ = $\emptyset$ for each class X. After R1-1 has been applied, the map function generates the key-value pair $\langle A$,$A \sqcap B \sqsubseteq C\rangle$. Other pairs such as $\langle X$, $S(X)\rangle$ and $\langle \top$, $S(X)\rangle$ for each X are produced too. The intermediate output is used in the reduce function. The results is that for key A, $A \in S(A)$ and $A \sqcap B \sqsubseteq$ C, (B, C) is added to $P(A)$. After adding the new axiom to the set of existing axioms, all the axioms

are executed in the next step. The output of the map phase for applying R1-2 are the following key-value pairs:

$$\{\langle A, A \in S(A)\rangle, \langle B, (B,C) \in P(A)\rangle, \langle A, A \sqsubseteq B\rangle, \langle A, A \sqsubseteq D\rangle\}$$

In the reduce phase, since both $A \in S(A)$ and $A \sqsubseteq B$ are associated with key A, B can be added to $S(A)$. Applying the R1-2 rule, C and D are added to $S(A)$. When the map function is executed, since B is in $S(A)$, the pair $\langle B, B \in S(A)\rangle$ and $\langle B, (B, C) \in P(A)\rangle$ will be generated. In the reduce phase, both tuples have the same key B. Using the conjunction rule, C is added to $S(A)$.

3.4.2 ELK

ELK is a Java-based specialized reasoner for OWL EL ontologies by using multiple cores/processors to speed up the reasoning process. Since the first release version of ELK, it has been widely used in a variety of application areas, such as biology and medicine, which requires efficient reasoners to handle large biomedical ontologies [30].

System Module

The main software modules of ELK are shown in Figure 3.8. The direction of arrows indicates the information flow during classification. There are two independent entry points: the Command-line Client and the Protégé Plugin to the left. The Command-line Client extracts OWL ontologies from files in OWL Functional Style Syntax (FSS). The Protégé Plugin is applied the ELK's bindings to OWL API[1] to get data from Protégé[2]. The ELK reasoner is divided into three packages. The *standalone client* includes the command-line client and the FSS parser for reading OWL ontologies. The *Protégé plugin* allows ELK to be used as a reasoner in Protégé and compatible tools. The *OWL*

[1]OWL API is available at http://owlapi.sourceforge.net/
[2]Protégé is available at http://protege.stanford.edu/

API bindings package allows ELK to be used as a software library which is controlled via the OWL API interfaces. The next step is based on ELK's representation of OWL objects (axioms and expressions) instead of the OWL API.

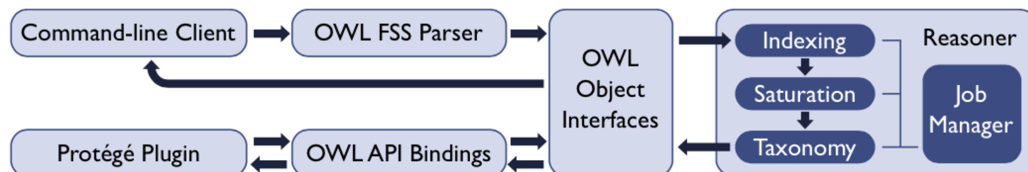


FIGURE 3.8: Main modules of ELK and information flow during classification (adapted from [30])

Reasoning Algorithm

The ELK reasoning component works by deriving consequences of ontology axioms under inference rules. The main components of the core reasoning algorithms implemented in ELK can be divided into three phases: indexing, saturation and taxonomy construction [29].

Indexing The indexing phase builds data structures which can be used to effectively check the conditions of the inference rules and the index assigned to concepts and roles occurring in the given ontology. In ELK, indexing is executed in a second thread in parallel to loading the ontologies. In addition, ELK keeps record of the exact counts of negative and positive occurrences of concepts to incrementally update the index structure without reloading the whole ontology.

Saturation The saturation phase computes the deductive closure of the input axioms following the inference rules. The optimization of this phase can affect the overall efficiency. The algorithm (see Algorithms 4+5) maintains two collections of axioms: the set of *processed axioms* for which the rules have been applied and the *scheduled*

queue of the remaining axioms. The algorithm works repeatedly to pop up an axiom from the *scheduled queue*. If an axiom is not in the processed set, it is moved to this set and all inferences resulting from this axiom and the processed axiom are added to the end of the queue. During processing, if all the processors execute concurrently, there may exist conflicts among the threads. In order to solve this problem, ELK uses a lock-free technique to distribute the axioms according to a 'context' in which the axioms can be used as premises of inference rules and processed independently. It is an active context if the scheduled queue of this context is not empty. For every input axiom, the algorithm adds every context assigned to this axiom and this axiom to the queue of the scheduled axioms for this context. If the queue of scheduled axioms is non-empty, the context is activated and added to the active contexts. Each active context is repeatedly processed in a loop.

Algorithm 4: activeContexts.activate (context) [30]

1 **if not** context.isActive **then**
2   context.isActive $\leftarrow$ **true**;
3   activeContexts.put (context);

Algorithm 5: activeContexts.deactivate (context) [30]

1 context.isActive $\leftarrow$ **false**;
2 **if** context.scheduled $\neq \emptyset$ **then**
3   activeContexts.activate (context);

Taxonomy Construction Taxonomy construction is the output of the classification which only contains direct subsumptions representing equivalence classes of concepts, such that if a taxonomy contains $A \sqsubseteq B$ and $B \sqsubseteq C$ then $A \sqsubseteq C$ should not exist,

unless some of them are equivalent. Therefore, the computation of subsumption relationships between concepts must be transitively reduced. Due to the fact that the number of all predecessors of a concept A is sizeable and the number of direct predecessors is countable, it implements the Transitive Reduction method performing the inner iteration only over the set of direct predecessors of A which have been found. For the given concept A, the method computes two sets A.equivalentConcepts and A.directPredecessors. The former set contains all the concepts which are equivalent to A including itself. The latter set contains exactly one concept from each equivalence class of direct predecessors of A. For multiple concepts, they execute independently on parallel processors. According to the computation of two sets for each concept, one taxonomy concept for each distinct class of equivalent concepts is constructed and connects concepts based on their direct predecessors relationships. At last, top ($\top$) and bottom ($\bot$) are added in the proper positions of the ontology.

3.4.3 Snorocket

Snorocket implements a concurrent classification algorithm which allows using synchronous processing in multi-processor machines and supports concrete domains.

Implementation

The implementation of the current version Snorocket is targeted at supporting the OWL $\mathcal{EL}$ profile. In DLs, a concrete domain can be used to define new classes by specifying restrictions on attributes that have literal values. For example, considering the following axioms:

$$toddler \equiv person \sqcap \exists hasAge.(\leq, 3)$$

$$child \equiv person \sqcap \exists hasAge.(\leq, 17)$$

After the normalization, these axioms can be transferred into the following axioms:

$$\exists hasAge.(\leq,17) \sqsubseteq A \qquad person \sqcap A \sqsubseteq child$$

$$child \sqsubseteq person \qquad child \sqsubseteq \exists hasAge.(\leq,17)$$

$$\exists hasAge.(\leq,3) \sqsubseteq B \qquad person \sqcap B \sqsubseteq toddler$$

$$toddler \sqsubseteq person \qquad toddler \sqsubseteq \exists hasAge.(\leq,3)$$

From these axioms we can infer that a toddler is also a child, but a child may not a toddler, when analyzing the expressions $toddler \sqsubseteq \exists hasAge.(\leq,3)$ and $\exists hasAge.(\leq ,17) \sqsubseteq A$. After comparison with the arguments $(\leq,3)$ and $(\leq,7)$, the result returns a positive match if all the possible values of the first operator-pair are covered by the possible values of the second operator-value pair. Otherwise, the result returns false. Moreover, the binary operators $<$, $<=$, $>$, $>=$ can also be used in a concrete domain expression and attributes can have other types of values.

The new version of Snorocket implements a multi-threaded saturation algorithm which is inspired by the saturation method used in ELK (see Section 5.2.2). The core part for this method is to split the computation into small partitions which can be processed by each worker independently and concurrently to reduce the locking overhead during the classification [29, 38].

3.4.4 Konclude

Konclude is a DL reasoner, which incorporates different reasoning procedures and implements new as well as extensions of existing optimizations to support a multi-core, shared memory system.

Architecture

The Konclude system provides two kinds of communication: one is an OWL link server that exposes ontology management and reasoner functionality to other clients; the other interacts with the reasoner via a command line interface, which can load an

ontology, execute a basic reasoning request with a system configuration. The overall workflow for handling ontologies and reasoning requests can be divided into three steps: parsing, loading and reasoning. For each processing step, there is a manager for controlling the execution [53].

Konclude can concurrently handle several ontologies. It is necessary for the system to answer a request for a certain ontology. Therefore, in the parsing step, axioms of an ontology are first collected in containers to keep track of the different revisions of an ontology. The reasoning manager plays an important role in handling the requests that require reasoning. In order to generate an answer, the requests are characterized by a list of conditions that have to be satisfied. Then the reasoning manager identifies and manages the process to satisfy the conditions of these requests. For example, if the user requests the class hierarchy of an ontology, it is necessary to build the internal presentation and data structure, test the consistency and classify the ontology.

Optimization

Parallel ontology processing is one of the main feature for the Konclude system. It can classify several ontologies concurrently and divide all the tasks into independent threads. This is especially useful when Knoclude uses an OWL link sever to serve multiple clients that operate on different servers. Furthermore, using peer-to-peer messages for communication can avoid conflicts and starvation of the execution system. However, the TBox classification of Konclude implemented with sequential not parallel methods.

Evaluation

The experimental results from [29, 38, 53] provide an evaluation that compares the

reasoners Konclude, FaCT++, HermiT, and Pellet [3], Snorocket and ELK for $\mathcal{EL}$ ontologies.

First, in order to facilitate a comparison between the reasoners that is independent of the number of CPU cores, the experiments compare the parallelized reasoners ELK and Konclude with only one worker thread and separately evaluated the effect of parallelization.

Second, the experiments compare the average classification time for different reasoner in different ontologies especially some large ontologies including SNOMED CT[4], FMA-lite[5] and OWL $\mathcal{EL}$ version of GALEN[6].

The result shows that Konclude performs well on small ontologies using only one thread when compared with ELK, which has a good performance on many larger $\mathcal{EL}$ ontologies. Both ELK and Snorocket outperformed the other reasoners on largel tested ontologies. The average classification time of ELK is 2-3 times faster than Snorocket. When the ontology consists of n disjoint and equal components that can be classified independently, the average classification time of ELK can be significantly reduced when computing the same results compared with other reasoners. Therefore, it is promising to apply optimized concurrent techniques to reduce the classification time of large complex ontologies.

3.5 Summary

In this chapter, we presented different classification methods and three popular DL reasoners, which apply optimization techniques in various different ways. For the sequential methods, all of them focus on reducing the number of subsumption tests

[3]Pellet is available at http://pellet.owldl.com/
[4]SNOMED CT is available at http://ihtsdo.org/
[5]FMA-lite is available at http://www.bioontology.org/wiki/index.php/FMAInOwl
[6]GALEN is available at http://condor-reasoner.googlecode.com

and improving time complexity to get a better performance. For the parallel methods, considering two important factors: number of threads and partition size, both of them play important roles during the whole procedure. The parallel reasoners we introduced implement different parallel techniques to improve the efficiency of reasoners. However, most of them do not focus on TBox classification and their efficiency can be affected by updating information and precomputing among different threads. Therefore, when we design parallel TBox classification methods, it is necessary to consider the impact factors and find efficient solutions to improve the performance of TBox classification. In the next chapter, based on these existing methods, a new parallel TBox classification approach is introduced, which is inspired by the existing methods and improves speed up factors by applying novel parallel strategies.

4 Parallel Reasoning

4.1 Introduction

In this chapter, the first version parallel framework is motivated by previous parallel approaches and also expands ideas presented in [21] to parallel processing. This HPC approach is implemented with a shared-memory architecture, atomic global data structures, and new strategies for parallel subsumption testing, which is ideally suited for shared-memory SMP servers, but does not rely on locking techniques and thus avoids possible race conditions. Specifically, it is mainly focused on the differences and novelties to speed up the OWL classification process: using parallel processing [31], with hundreds of threads, in combination with an atomic multi-dimensional data structure is shared among a pool of processors performing pre-computation and classification in parallel. Compared to [3], where a small set of threads operated on a shared taxonomy via locking, this architecture can update subsumption relations lock-free in a globally shared taxonomy. In comparison to [62] this architecture avoids a multitude of subsumption tests due to shared data.

4.2 Architecture

The goal of this approach is to parallelize the computation of subsumption taxonomies consisting of a large number of concepts and speed up the process of TBox classification. In order to reuse information from (non-)subsumption tests, this method implements a parallel framework and a shared-memory global data structure to record all binary subsumption relationships occurring in an ontology $\mathcal{O}$ (or TBox). A set P contains all *possible* subsumees that every concept could have and a set K represents all subsumees found from *known* subsumption relationships or subsumption tests. For example, if $\mathcal{O}$ entails $B \sqsubseteq A$ (denoted as $\mathcal{O} \models B \sqsubseteq A$), then B is inserted into K_A and delete B from P_A. Since the classification of $\mathcal{O}$ tests all pairs of concept subsumptions, the concepts remaining in possible subsumee sets is used to reflect the amount of work that still needs to be done until P becomes empty. The predicate $subs?()$ is used to test subsumption relationships for each pair of concepts in P. The call of $subs?(B, A)$ returns true if B subsumes A and false otherwise. Before testing, it is necessary to know the satisfiability of each concept, e.g., by testing $subs?(\bot, A)$.

In an ontology $\mathcal{O}$, a set $N_{\mathcal{O}}$ contains all concepts occurring in $\mathcal{O}$. For each concept $X \in N_{\mathcal{O}}$, the method initializes P_X, which contains all possible subsumees of X and an initially empty K_X to contain all the known subsumees derived from subsumption tests. For instance, let us assume three concepts $\{A, B, C\} \subseteq N_{\mathcal{O}}$. After initialization, $P_A = \{B, C\}$, $P_B = \{A, C\}$, $P_C = \{A, B\}$ and $K_A = K_B = K_C = \emptyset$. Since $N_{\mathcal{O}}$ contains all concepts from $\mathcal{O}$, in the following phases $N_{\mathcal{O}}$ is used as a global parameter for classifying $\mathcal{O}$ in parallel. For example, for the concepts A, B and C, subsumption (and indirectly satisfiability) for the pairs below are computed using $subs?()$: $\{\langle \bot, C \rangle, \langle A, C \rangle\}$, $\{\langle \bot, B \rangle, \langle C, B \rangle\}$. The results are $\mathcal{O} \models C \sqsubseteq A$ and $\mathcal{O} \models B \not\sqsubseteq C$. The changes to P and K are : P_A = {B, ~~C~~}, P_C = {A, ~~B~~}, and $K_A = \{C\}$. In order to guarantee the soundness and completeness of this algorithm, a complete possible set for each concept is created in

$N_{\mathcal{O}}$ before any possible subsumees could be removed from P. In addition, a set $R_{\mathcal{O}}$ is used for containing each concept $X \in N_{\mathcal{O}}$ where $P_X \neq \emptyset$.

Algorithm 6: parallelTBoxClassification(P, K)

1 **Input:** P, K - sets of *possible* and *known* subsumees
2 **Output:** $\mathcal{H}$ - the whole ontology taxonomy
3 $N_{\mathcal{O}} \leftarrow generateNodeSet(\mathcal{O})$
4 $T \leftarrow createWorkerPool()$
5 $L_{\mathcal{O}} \leftarrow getRandomOrder(N_{\mathcal{O}})$
6 $G \leftarrow randomDivision(L_{\mathcal{O}})$
7 **for each** *group* $G_i \in G$ **do**
8   **if** $getAvailableThread(T)$ **then**
9         $randomDivisionSubsTest(G_i)$
10 $R_{\mathcal{O}} \leftarrow generateRemainingPossibleSet()$
11 $G \leftarrow groupDivision(R_{\mathcal{O}})$
12 **while** $R_{\mathcal{O}} \neq \emptyset$ **do**
13   **for each** *group* $G_X \in G$ **do**
14   **if** $getAvailableThread(T)$ **then**
15             $groupDivisionSubsTest(G_X)$
16 $X \leftarrow computeTopConcept()$
17 **while** $K_X \neq \emptyset$ **do**
18   **if** $getAvailableThread(T)$ **then**
19         $\mathcal{H}_X \leftarrow buildPartialHierarchy(K_X)$
20   **if** $\mathcal{H}_X \neq \emptyset$ **then**
21         $\mathcal{H} \leftarrow buildOntologyTaxonomy(\mathcal{H}_X)$
22     $X \leftarrow getKnownSubsumees(K_X)$
23 **return** $\mathcal{H}$

The TBox classification process is implemented in three parallel phases. In each phase different parallelization strategies are applied. A global parameter w is used to specify the maximum number of parallel threads (or workers) available for classification. The architecture of this approach is shown in Figure 4.1 and the complete algorithm

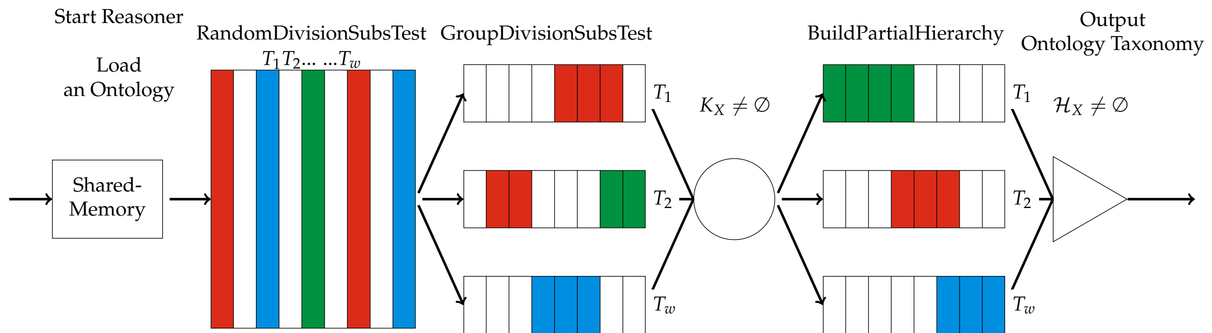


FIGURE 4.1: The Architecture of Parallel TBox Classification Approach

of parallelTBoxClassification(P, K) is shown in Algorithm 6. In the first phase, the method randomly partition the set of all named concepts into disjoint sequences having almost identical sizes obtained by dividing the total number of named concepts by w (see line 7-9). In the second phase, all concepts X with $P_X \neq \emptyset$ are found using a group division strategy with *round-robin scheduling* for the worker thread pool in order to finish the classification process (see line 10-15). In the final phase, a parallel *divide-and-conquer* framework is applied. Partial hierarchies are generated in the divide part for all concepts X with $K_X \neq \emptyset$. In the conquer part the whole ontology is constructed based on the existing partial hierarchies where $\mathcal{H}_X \neq \emptyset$ (see line 16-22).

4.3 Ontology Classification

In the classification phase, two strategies are designed, the random and the group division strategy. In this algorithm, each concept has a global set which contains *possible* (P) and *known* subsumees (K). In that way the changes are updated in the global sets during classification. Each thread tests subsumption relationships and removes as many concepts from P as possible. TBox classification terminates once P has become empty for all concepts in $N_{\mathcal{O}}$.

Definition 4.1 With reference to $N_{\mathcal{O}}$, the set $R_{\mathcal{O}} = \bigcup_{X \in N_{\mathcal{O}}} P_X$ contains all remaining possible subsumees P_X of each concept X.

4.3.1 Random Division Strategy

According to the number of threads and total number of concepts occurring in $\mathcal{O}$, all concepts are divided into different groups with almost the same size. In order to make the best use of all idle threads, the number of threads is identical to the number of groups for testing subsumption relationships for all concepts in $N_{\mathcal{O}}$. The method first generates an unordered sequence $L_{\mathcal{O}}$ which includes all concepts. Then $L_{\mathcal{O}}$ is divided into w different groups, where w is the number of available threads. Then subsumption relationships are tested between all pairs $\langle Y, X \rangle$ with $Y, X \in N_{\mathcal{O}}$ for each group G_i by calling randomDivisionSubsTest(G_i) (see Algorithm 7). The $sat?()$ is used to test concept satisfiability and $tested()$ to check whether the subsumption between two concepts has already been tested.

Example 4.1 Assume there are three threads available to perform subsumption tests. The algorithm first shuffles all concepts in $N_{\mathcal{O}} = \{A, B, C, D, E, F\}$ and returns the first cycle sequence $L^1_{\mathcal{O}} = (A, C, E, D, B, F)$. Then each group G_i contains two possible subsumees, such as $G_1 = \{A, C\}$, $G_2 = \{E, D\}$, and $G_3 = \{B, F\}$ for subsumption testing. For each thread T_i the results are: $T_1 : C \sqsubseteq A;\ T_2 : D \not\sqsubseteq E;\ T_3 : F \not\sqsubseteq B$.

Algorithm 7: randomDivisionSubsTest(G_i)

1 **Input:** G_i - random division group
2 **Output:** K - sets of known subsumees
3 P - sets of remaining possible subsumees
4 **for each** *concept pair* $\langle X, Y \rangle \in G_i$ **do**
5   **if** $\neg tested(X, Y)$ **then**
6     $satX \leftarrow sat?(X)$
7     $satY \leftarrow sat?(Y)$
8   **if** $\neg satX$ **then**
9       $P_X \leftarrow \emptyset$
10   delete X from P_Y
11   **else if** $\neg satY$ **then**
12       $P_Y \leftarrow \emptyset$
13   delete Y from P_X
14   **else**
15   **if** $subs?(X, Y)$ **then**
16   insert Y into K_X
17   delete Y from P_X

The second cycle sequence is $L^2_{\mathcal{O}} = (C, D, A, F, B, E)$. The divisions of each group are $G_1 = \{C, D\}$, $G_2 = \{A, F\}$ and $G_3 = \{B, E\}$. For each thread, the results are : $T_1 : D \sqsubseteq C$; $T_2 : F \sqsubseteq A$; $T_3 : E \sqsubseteq B$.

Therefore, after applying the changes to P and K, the results are as follows:

$$P_A = \{B, \cancel{C}, D, E, \cancel{F}\} \quad K_A = \{C, F\}$$

$$P_B = \{A, C, D, \cancel{E}, \cancel{F}\} \quad K_B = \{E\}$$

$$P_C = \{A, B, \cancel{D}, E, F\} \quad K_C = \{D\}$$

$$P_D = \{A, B, C, E, F\} \quad K_D = \emptyset$$

$$P_E = \{A, B, C, \cancel{D}, F\} \quad K_E = \emptyset$$

$$P_F = \{A, B, C, D, E\} \quad K_F = \emptyset$$

Since the process to generate random divisions currently ignores already discovered subsumptions, there is a possibility that a pair of concepts occurs in a division more than once in different cycles. Therefore, the *tested*() is used to avoid redundant tests. The runtime for each thread are considered almost the same and the waiting time can be neglected right now. Currently, the results also show that the runtime differences for each thread can be neglected when compared with the total execution time.

If $R_{\mathcal{O}}$ is not empty after random division phase testing, possible subsumees are left in P. A group division strategy is designed to divide all remaining possible subsumees in $R_{\mathcal{O}}$ into different groups to continue testing subsumption relationships until P becomes empty.

4.3.2 Group Division Strategy

For each concept X in $N_{\mathcal{O}}$ a group $G_X = P_X$ is generated according to the remaining set $R_{\mathcal{O}}$ which is defined in Definition 1. The groups G_X define the input to groupDivisionSubsTest(G_X) (see Algorithm 8), which determines what elements of G_X are subsumed by X. Each group is assigned to a different idle thread until all groups have been classified. During the process, *round-robin scheduling* is applied to ensure a good use of all threads.

Algorithm 8: groupDivisionSubsTest(G_X)

1 **Input:** G_X - group division of concept X
2 **Output:** K - sets of known subsumees
3 P - sets of remaining possible subsumees
4 **for each** *concept* $Y \in G_X$ **do**
5 **if** $sat?(Y)$ *and* $\neg tested(X, Y)$ **then**
6 **if** $subs?(X, Y)$ **then**
7 insert Y into K_X
8 delete Y from P_X

Example 4.2 According to the results from the random division phase (see Example 4.1), let us assume the following six groups are generated:

$$G_A = \{B, D, E\}$$
$$G_B = \{A, C, D\}$$
$$G_C = \{A, B, E, F\}$$
$$G_D = \{A, B, C, E, F\}$$
$$G_E = \{A, B, C, F\}$$
$$G_F = \{A, B, C, D, E\}$$

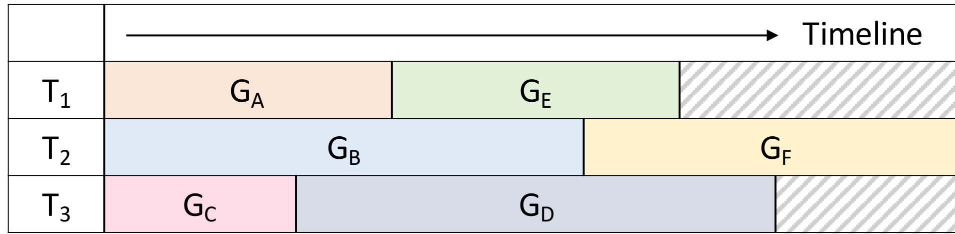


FIGURE 4.2: Scheduling results for Example 3.2

In the following we assume all concepts are satisfiable, the group scheduling is shown in Figure 4.2 and the results of each thread are shown as follows.

$$T_1(G_A) : B \sqsubseteq A, D \sqsubseteq A, E \sqsubseteq A;$$

$$T_2(G_B) : A \not\sqsubseteq B, C \not\sqsubseteq B, D \not\sqsubseteq B;$$

$$T_3(G_C) : A \not\sqsubseteq C, B \not\sqsubseteq C, E \not\sqsubseteq C, F \sqsubseteq C;$$

$$T_3'(G_D) : A \not\sqsubseteq D, B \not\sqsubseteq D, C \not\sqsubseteq D, E \not\sqsubseteq D, F \not\sqsubseteq D;$$

$$T_1'(G_E) : A \not\sqsubseteq E, B \not\sqsubseteq E, C \not\sqsubseteq E, F \not\sqsubseteq E;$$

$$T_2'(G_F) : A \not\sqsubseteq F, B \not\sqsubseteq F, C \not\sqsubseteq F, D \not\sqsubseteq F, E \not\sqsubseteq F;$$

Since P becomes empty and $R_{\mathcal{O}} = \emptyset$, all subsumption relationships between all concepts occurring in $\mathcal{O}$ have been tested. The classification of $\mathcal{O}$ terminates.

4.3.3 Ontology Taxonomy

In order to find the direct subsumees of each concept and build the whole subsumption hierarchy, a concept hierarchy strategy is applied which is implemented by a divide-and-conquer algorithm to construct the taxonomy of $\mathcal{O}$. When $R_{\mathcal{O}}$ becomes empty, all known subsumees of a concept X are members of K_X. First, find the top concept A and traverse all the concepts $X \in K_A$. Then the partial hierarchy $\mathcal{H}_{\mathcal{X}}$ is built for each

concept X by computing the transitive closure to reduce the known set K_X. For each concept in K_X, we compute all the direct subsumees of X and insert them into $\mathcal{H}_\mathcal{X}$. Finally, the whole taxonomy of the ontology $\mathcal{O}$ is constructed based on the partial hierarchy of each concept.

Concept Hierarchy Strategy

In the divide phase, the algorithm begins with K_X where X is initially equal to $\top$. For each concept $Y_i \in K_X$ and $i = 1, 2...n$, if $K_{Y_i} \neq \emptyset$ and $X \in K_{Y_i}$, then $Y_i \equiv X$; if $X \notin K_{Y_i}$, $Z_i \in K_{Y_i}$ and $Z_i \in K_X$, then Z_i is deleted from K_X. The method continues with the next concept $Y_{i+1} \in K_X$ until all the concepts in K_X have been traversed. The remaining concepts in K_X are the direct subsumees of X which are inserted into $\mathcal{H}_\mathcal{X}$. The algorithm buildPartialHierarchy(K_X) is shown in Algorithm 9. For each concept X with $K_X \neq \emptyset$ its partial hierarchy is built in parallel. The process terminates once all partial hierarchies have been built.

Algorithm 9: buildPartialHierarchy(K_X)

```
1  Input: K_X – set of known subsumees of concept X
2  Output: H_X – the partial hierarchy of concept X
3  if K_X ≠ ∅ then
4      for each concept Y ∈ K_X do
5          if K_Y ≠ ∅ then
6              if X ∈ K_Y then
7                  delete X from K_Y
8                  setEquivalentConcept(X, Y)
9              else
10                 for each concept Z ∈ K_Y do
11                     if Z ∈ K_X then
12                         delete Z from K_X
13     H_X ← K_X
14 return H_X
```

Example 4.3 According to the results from Example 4.2, when P becomes empty, the known sets for each concept are:

$$K_A = \{B, C, D, E, F\}$$

$$K_B = \{E\}$$

$$K_C = \{D, F\}$$

$$K_D = \emptyset$$

$$K_E = \emptyset$$

$$K_F = \emptyset$$

Since $A \equiv \top$, the hierarchy construction starts with the first concept $B \in K_A$ and E is the first concept in K_B which is also in K_A, then E is deleted from K_A. The second concept is $C \in K_A$ and there are two concepts D and F in K_C, then D and F are deleted

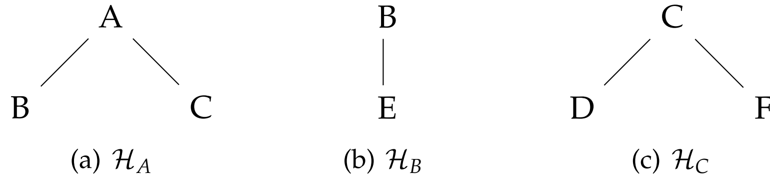


FIGURE 4.3: Partial Hierarchy for the concepts in different threads

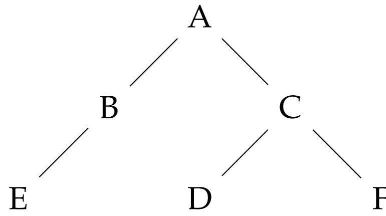


FIGURE 4.4: The whole concept hierarchy of $\mathcal{O}$

from K_A. Therefore $K_A = \{B, C\}$ and the partial hierarchy of A is $\mathcal{H}_A = \{B, C\}$. Since $K_B = \{E\}$ and $K_E = \emptyset$, the partial hierarchy of B is $\mathcal{H}_B = \{E\}$. Because of $K_C = \{D, F\}$, $K_D = \emptyset$ and $K_F = \emptyset$, the partial hierarchy of C is $\mathcal{H}_C = \{D, F\}$. The final partial hierarchy $\mathcal{H}$ of the concepts in each thread is as follows:

$$T_1 : \mathcal{H}_A = \{B, C\} \text{ in Figure 4.3(a)}$$

$$T_2 : \mathcal{H}_B = \{E\} \text{ in Figure 4.3(b)}$$

$$T_3 : \mathcal{H}_C = \{D, F\} \text{ in Figure 4.3(c)}$$

In the conquer phase, after the partial hierarchy of each concept has been built, all the partial hierarchies are merged into the whole taxonomy from top to bottom. The final concept hierarchy of $\mathcal{O}$ is shown in Figure 4.4.

4.4 Optimization

Due to the possibly large size of ontologies and the cost of subsumption tests, we propose a modified half-matrix data structure that uses less memory and requires less computation and also apply an improved group division strategy (see Section 4.4.2) to get a better performance especially for some complex ontologies. More subsumption relationships can be inferred by applying transitive closure without subsumption tests.

4.4.1 Half-Matrix Structure

In order to remove potential non-possible or known subsumees from the *Possible* list, this algorithm uses a half-matrix to represent all possible relations for each concept. If a concept C from $\mathcal{O}$ is satisfiable, mark it with a unique index I_C. Each concept A with a smaller index I_A contains the possible relationships with concept B with a bigger index in P_A. Therefore the set P contains all possible relationships which could be possible subsumers or subsumees. For each concept its known set contains all its subsumees. Possible relations for each pair of concepts are only represented once. For instance, suppose that $C \not\sqsubseteq A$ and $A \not\sqsubseteq C$, if the index of the possible subsumee C is bigger than the index of the current concept A, then delete C from P_A; otherwise delete A from the possible set P_C.

This algorithm computes subsumption tests symmetrically for every pair of concepts. Assume the ontology $\mathcal{O}$ computes the pairs $\langle C, A \rangle$ and $\langle F, B \rangle$, then $subs?(A, C)$, $subs?(C, A)$ and $subs?(B, F)$, $subs?(F, B)$ are tested. The results are $\mathcal{O} \models C \sqsubseteq A$, $\mathcal{O} \models A \not\sqsubseteq C$ and $\mathcal{O} \models F \not\sqsubseteq B$, $\mathcal{O} \models B \not\sqsubseteq F$. Using the half-matrix, since $I_A < I_B < I_C < I_D < I_E < I_F$, the changes to P and K result in the following sets:

$$
\begin{array}{lll}
I_A = 1 & P_A = \{B, \not{C}, D, E, F\} & K_A = \{C\} \\
I_B = 2 & P_B = \{C, D, E, \not{F}\} & K_B = \emptyset \\
I_C = 3 & P_C = \{D, E, F\} & K_C = \emptyset \\
I_D = 4 & P_D = \{E, F\} & K_D = \emptyset \\
I_E = 5 & P_E = \{F\} & K_E = \emptyset \\
I_F = 6 & P_F = \emptyset & K_F = \emptyset
\end{array}
$$

Therefore, there are two results from testing the relations between every pair of concepts. This ensures that there will be changes in P and K for every two symmetrical tests until all concepts in P have been tested.

4.4.2 Improved Division Strategy

In the Group Division Phase (see Section 4.3.2) *round-robin scheduling* is applied. However, in the tests we encountered some difficult ontologies where the runtime of subsumption tests is not uniform, especially some ontologies with QCRs. The division strategy from Section 4.3.2 does not have a specific solution for this kind of ontologies. In Example 4.2, a queue $Q = \{G_A, G_B, G_C, G_D, G_E, G_F\}$ of pending tasks is created. Suppose only three threads are available and each thread receives a task from Q. When a task is finished, an idle thread gets another task assigned based on the sequence of tasks in Q. However, one can observe that when the second set of tasks ($G_{D,} G_E$) is finished for T_1 and T_3, T_2 is still working on G_F and, thus, leaves threads T_1 and T_3 idle until classification terminates (see Figure 4.2).

To improve the performance of this method and ensure a more efficient use of multiple threads for these difficult ontologies, the *Fork/Join framework* is applied for the improved group division strategy. Currently, this strategy to divide a task into smaller subtasks depends on the size of the ontology and the number of available threads. If a task is small enough, which means based on previous results the expected runtime

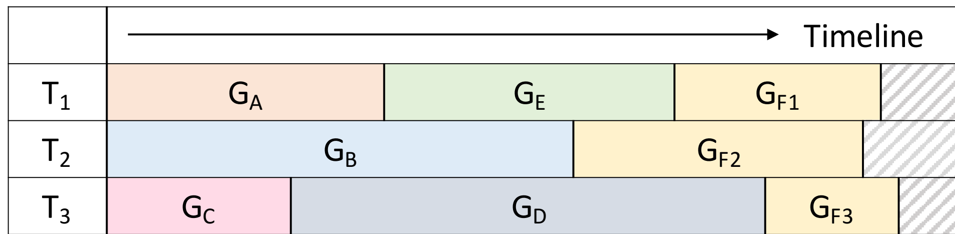


FIGURE 4.5: Improved scheduling results for Example 4.2

for a particular group is much smaller than the idle time of threads, the improved algorithm will execute the subsumption tests for this group directly without task division. Otherwise, the task for the group which includes difficult subsumption tests will be divided according to the number of available threads. In the case of Example 4.2, since there are three threads available, the task G_F will be divided into three subtasks G_{F1}, G_{F2}, G_{F3} that are added to sub-queue Q_F.

During execution, if idle threads are waiting in the thread pool, the *work stealing strategy* is applied to steal tasks from other threads that are still busy using the sub-queues created for each concept. Accordingly, the subtasks G_{F1}, G_{F2}, G_{F3} are assigned to idle threads (see Figure 4.5). Although we cannot guarantee that all the threads will finish at the same time, the runtimes and speedup factors have been improved, especially for some difficult ontologies, and the overhead has been significantly reduced. Therefore, both the total running time for the Group Division Phase and the waiting time of idle threads can be improved by applying the improved group division strategy. The algorithm improveScheduling(Q) is described in Algorithm 10.

Algorithm 10: improveScheduling(Q)

```
1  Input: Q - A queue of unclassified groups
2  Output: K - sets of known subsumees
3  while ¬isEmpty(Q) do
4      G_i ← deQueue(Q)
5      while T_i ← getAvailableThread(T) do
6          T_i → groupDivisionSubsTest(G_i)
7  for each thread T_i ∈ T do
8      if T_i is busy with G_j then
9          for each sub-group G_{j_k} do
10             G_{j_k} ← splitSubtask(G_j)
11             add G_{j_k} to sub-queue Q_s
12         improveScheduling(Q_s)
```

4.4.3 Optimized Parallel Phase

In order to shrink the set P by using less subsumption tests, known results from subsumption tests are used to prune untested possible concepts in P without subsumption testing. Given the results from Example 4.2, assume concept $B \in P_A$ will be tested for a subsumption relationship with A. The following steps perform changes to P and K before new divisions are created for an idle thread.

Situation 1 If both concepts are unsatisfiable, their set P is empty; The changes to P and K are $P_A = \emptyset$, $P_B = \emptyset$, $K_A = \emptyset$ and $K_B = \emptyset$.

Situation 2 If both concepts are satisfiable, test the subsumption relationships between them.

Definition 4.2 If the index of A is smaller than B, i.e., $I_A < I_B$, the position of concept B in P_A is defined as: $B.position = P_A.position[I_B - I_A - 1]$.

Situation 2.1 If concept $B \in P_A$ and $tested(A, B)$ is true, which means B has been tested, then we continue with the next concept $C \in P_A$ to test its subsumption relationships with A; otherwise continue with Situation 2.2.

Situation 2.2 The subsumption relationships are tested in a symmetrical way by $subs?(B, A)$ and $subs?(A, B)$. If both results are true, then the two concepts are equivalent to each other; otherwise continue with Situation 2.3.

Situation 2.3 If only one of the results is true, i.e., $\mathcal{O} \models B \sqsubseteq A$ but $\mathcal{O} \models A \not\sqsubseteq B$, the changes to both sets P and K are P_A = {$\not{B}$, $\not{C}$, D, E, $\not{F}$}, $K_A = \{B, C, F\}$ and we continue with Situation 2.3.1; otherwise continue with Situation 2.4.

Situation 2.3.1 Delete all concepts $Y \in K_B$ from P_A and K_A. Due to $\mathcal{O} \models B \sqsubseteq A$ and $K_B = \{E\}$, all the subsumees of B are subsumees of A but not the direct subsumee of A as shown in Figure 4.6. Therefore, all concept $Y \in K_B$ are deleted from P_A without subsumption tests. In the example, concept $E \in K_B$ but $E \notin K_A$ is deleted from P_A. The changes of P are P_A = {$\not{B}$, $\not{C}$, D, $\not{E}$, $\not{F}$} and we continue with Situation 2.3.2.

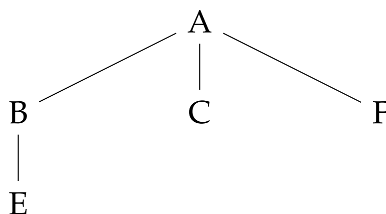


FIGURE 4.6: An Example for Situation 2.3.1 and 2.3.2

Situation 2.3.2 For all concepts $Y \in K_B$ delete A from P_Y. Due to $\mathcal{O} \models B \sqsubseteq A$ and $K_B = \{E\}$, all the subsumees of B are subsumees of A and concept A is not a subsumee of all concepts $Y \in K_B$ as shown in Figure 4.6. Therefore, concept A is deleted from

P_Y with $Y \in K_B$. Since only concept $E \in Y$ and $I_E > I_A$ in the example, there are no changes to both P and K.

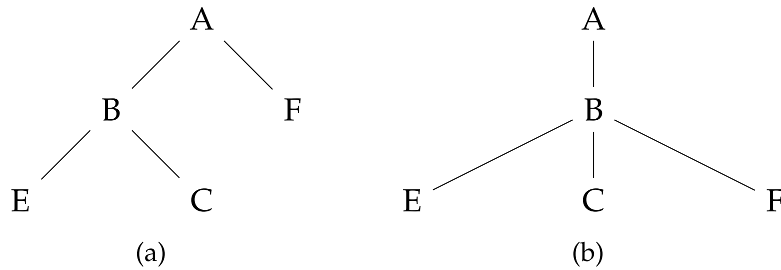


FIGURE 4.7: Counter examples for 'delete all concepts $X \in K_A$ from P_B'

We also consider situations such as 'delete all concepts $X \in K_A$ from P_B'. Since $K_A = \{B, C, F\}$, we know that concepts C and F are in K_A and the two concepts could not have subsumption relationships with B. However, there are some counter examples which indicate possible relationships between B and C, F such that $\mathcal{O} \models C \sqsubseteq B$ in Figure 4.7(a) and $\mathcal{O} \models F \sqsubseteq B$ in Figure 4.7(b). Therefore, we cannot assume subsumption relationships between $\langle B, C\rangle$ and $\langle F, B\rangle$ without performing subsumption tests.

Situation 2.4 If both concepts are not subsumed by each other such that $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, then both sets P and K remain unchanged.

According to this condition, we try to find some situations which allow us to shrink P in an efficient way without performing subsumption tests. However, we identified some counter examples as shown in Figure 4.9 where the dashed lines indicate possible relationships between pairs of concepts. Below we describe two scenarios.

- Delete all concepts $X \in K_A$ from P_B and $Y \in K_B$ from P_A. For example as shown in Figure 4.8, there is a concept $C \in K_A$, $C \in P_B$, $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but A and B are both known subsumers of C. The possible relationship between C and B could exist before C is deleted from P_B; there is a concept $E \in K_B$, $E \in P_A$,

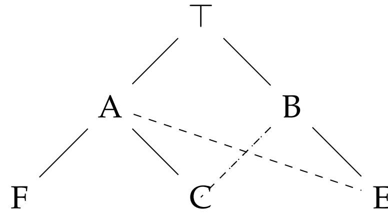


FIGURE 4.8: Counter Examples for Situation 2.4

$\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept E is a subsumee of both A and B. Therefore, the relationships of the pairs $\langle B, C \rangle$ and $\langle A, E \rangle$ need to be tested before deleting C from P_B and E from P_A.

- For all concepts $X \in K_A$ delete B from P_X and all concepts $Y \in K_B$ delete A from P_Y. In the example shown in Figure 4.9(a), there is a concept $F \in K_A$, $F \in P_B$ ($I_F > I_B$), $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept B is a known subsumee of F. In Figure 4.9(b), there is concept $E \in K_B$, $E \in P_B$, $\mathcal{O} \models A \not\sqsubseteq B$ and $\mathcal{O} \models B \not\sqsubseteq A$, but concept A is a subsumee of E. Therefore, relationships between the pairs of $\langle B, F \rangle$ and $\langle A, E \rangle$ need to be tested before deleting F from P_B ($I_B < I_F$) and E from P_A ($I_A < I_E$).

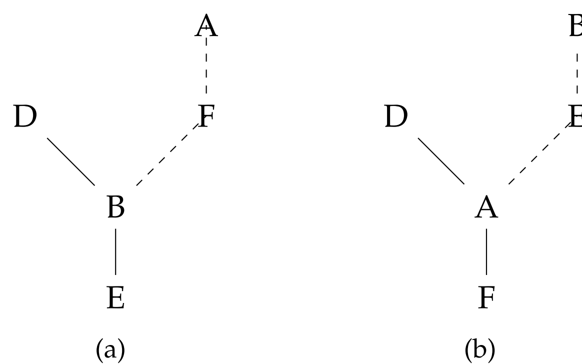


FIGURE 4.9: More Counter Examples for Situation 2.4

Algorithm 11 correctly deals with all the situations illustrated above.

Example 4.4 For random division tests, we apply the random division strategy and use the same random division results from Example 4.1. The first random division cycle results in:

$$T_1 : C \sqsubseteq A, A \not\sqsubseteq C;\ T_2 : E \not\sqsubseteq D, D \not\sqsubseteq E;\ T_3 : F \not\sqsubseteq B, B \not\sqsubseteq F$$

The results of the second random division cycle are:

$$T_1 : D \sqsubseteq C, C \not\sqsubseteq D;\ T_2 : F \sqsubseteq A, A \not\sqsubseteq F;\ T_3 : E \sqsubseteq B, B \not\sqsubseteq E$$

After finishing the random division tests, the changes to P and K result in:

$$\begin{array}{lll}
I_A = 1 & P_A = \{B, \not{C}, D, E, \not{F}\} & K_A = \{C, F\} \\
I_B = 2 & P_B = \{C, D, \not{E}, \not{F}\} & K_B = \{E\} \\
I_C = 3 & P_C = \{\not{D}, E, F\} & K_C = \{D\} \\
I_D = 4 & P_D = \{\not{E}, F\} & K_D = \emptyset \\
I_E = 5 & P_E = \{F\} & K_E = \emptyset \\
I_A = 6 & P_F = \emptyset & K_F = \emptyset
\end{array}$$

Algorithm 11: pruneNonPossible(A, B)

```
1  Input: A, B - two concepts from N_O
2  Output: K - sets of known subsumees
3          P - sets of possible subsumees
4  if sat?(A) then
5      if sat?(B) then
6          if ¬tested(B, A) and ¬tested(A, B) then
7              result_1 ← subs?(A, B)
8              result_2 ← subs?(B, A)
9              if result_1 and result_2 then
10                 return A ≡ B
11             else if result_1 then
12                 for each concept Y ∈ K_B do
13                     delete Y from P_A and K_A
14                     delete A from P_Y
15             else if result_2 then
16                 for each concept X ∈ K_A do
17                     delete X from P_B and K_B
18                     delete B from P_X
19     else
20         P_B ← ∅
21 else
22     P_A ← ∅
```

For each random division cycle, the above-mentioned optimized techniques are applied. Since $\mathcal{O} \models C \sqsubseteq A$ and concept $D \in K_C$, concept D is deleted from P_A and the remaining sets P become : P_A = {B, E}, P_B = {C, D}, P_C = {E, F}, P_D = {F}, P_E = {F}.

Now let us assume there are three threads available for subsumption testing and all concepts in $R_{\mathcal{O}}$ are divided into groups using the group division strategy. The divisions for the groups G_X are : G_A = {B, E}, G_B = {C, D}, G_C = {E, F}, G_D = {F}, G_E = {F}.

After applying the optimized techniques above for each thread T_i, all the pairs in brackets have not been tested and the results are:

$$T_1 : B \sqsubseteq A,\ A \not\sqsubseteq B\ (E \sqsubseteq A,\ A \not\sqsubseteq E);$$
$$T_2 : B \not\sqsubseteq C,\ C \not\sqsubseteq B\ (B \not\sqsubseteq D,\ D \not\sqsubseteq B);$$
$$T_3 : (E \not\sqsubseteq C,\ C \not\sqsubseteq E)\ F \sqsubseteq C,\ C \not\sqsubseteq F;$$
$$T_1' : D \not\sqsubseteq F,\ F \not\sqsubseteq D;$$
$$T_2' : (E \not\sqsubseteq F,\ F \not\sqsubseteq E);$$

For T_1, the subsumption relationship between concepts A and B is that $\mathcal{O} \models B \sqsubseteq A$, then concept $E \in K_B$ can be deleted from P_A without further testing by applying Situation 2.3.1. For T_2, the concepts B and C are not subsumed by each other and $D \in K_C$, then concept D can be deleted from P_B without further tests. For T_3, since the concepts B and C are not subsumed by each other and $E \in K_B$, concept E can be deleted from P_C without further tests. Since we use a global atomic data structure when testing the relationships between B and C, there will be no conflict between T_2 and T_3. The subsumption tests between C and E can be executed only after concepts B and C have been tested. For T_2', since concept $E \in K_B$, $F \in K_C$ and concepts B and C are not subsumed by each other, F can be deleted from P_E without further tests.

Therefore, all the subsumptions listed in brackets can be inferred without testing. The remaining possible set $R_{\mathcal{O}}$ will be pruned significantly due to the many relationships found among the concepts. The classification terminates when P has been emptied.

4.5 Summary

In this part, a novel parallel OWL ontology classification architecture has been presented. Different parallel techniques are applied to create a thread pool for each subtask working on an independent processor. Compared to existing sequential classification methods and the limitations of recently proposed parallel classification approaches, this method is the first in using a random division strategy to achieve a better scalability for ontologies of larger sizes and applying a group division strategy to finish TBox classification. Furthermore, due to the design of the shared atomic data structures possible race conditions are avoided for updates of shared data. The evaluation of the first version parallel method is presented in Chapter 6.

5 Improved Parallel Classification

5.1 Introduction

In this chapter, an improved parallel reasoning framework is proposed, which can be used to parallelize the classification process of OWL reasoners. Specifically, we mainly focus on three differences and novelties to speed up the OWL classification process: (i) An improved data structure is presented to adapt the information communication in different threads and reduce more potential relations among concepts applying transitivity closure. (ii) The adoption of *work-stealing* techniques [9, 18, 54] to manage adaptive and automatic load balancing for ontologies with varying degrees of reasoning complexity. Compared to the first version presented in Chapter 4 less memory and computation is required by avoiding overlaps among partitions, reducing the number of subsumption tests, and applying different parallelization techniques such as full-scale work stealing. (iii) The parallel reuse of major OWL reasoners as black-box subsumption testers. Compared to ELK [30, 29], this approach is more performant when many threads are used and is not restricted to a small subset of OWL. To the best of our knowledge, we are the first to propose a flexible parallel framework which can be applied to existing OWL reasoners in order to speed up their classification process.

5.2 Improved Data Structure

The goal of this method is to classify and construct the whole taxonomy and balance the allocation of resources and memory simultaneously in an efficient way. When it comes to parallelization, there are two important factors that affect the classification performance: concurrency and locking (waiting time). In order to balance these two problems with the potential occurrence of big-size ontologies and nonuniformity of subsumption tests, an atomic half-matrix shared-memory structure is created to maintain all the updated information with different sets and the parallel classification approach is mainly separated into two phases: precomputing (line 3-7) and classification phase (line 8-12) with black-box reasoners for each thread (line 22-25) in Algorithm 12.

5.2.1 Atomic Half-Matrix Structure $\mathcal{F}$

A shared-memory half-matrix structure $\mathcal{A}$ contains quadruples $\mathcal{A}_{\mathcal{C}_i}$ for each concept $\mathcal{C}_i \in N_{\mathcal{O}}$ with $N_{\mathcal{O}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ containing all satisfiable concepts of n ontology $\mathcal{O}$ (or TBox), n is the total number of concepts and $\mathcal{P}$ is a finite set of potential possible subsumees of all concepts in $N_{\mathcal{O}}$ (see line 15-19 in Algorithm 12). For all concepts $\mathcal{C}_i \in N_{\mathcal{O}}$, we use $\gtrdot$ to indicate an arbitrary but fixed order between every pair of concepts (line 20-21). For the pair $\langle \mathcal{C}_i, \mathcal{C}_j \rangle \in N_{\mathcal{O}}$, *if* $\mathcal{C}_i \gtrdot \mathcal{C}_j$, *then* all the operations related to $\mathcal{A}_{\mathcal{C}_i}$ and $\mathcal{A}_{\mathcal{C}_j}$ operate on the three sets $\mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i$ in $\mathcal{A}_{\mathcal{C}_i}$ with $\mathcal{A}_{\mathcal{C}_j}$ indexing $\mathcal{A}_{\mathcal{C}_i}$ and its related sets.

For all the satisfiable concepts in $N_{\mathcal{O}}$, a half-matrix structure represents all possible relations with other concepts inferred or tested by a black-box reasoner, e.g., $\textsc{subs?}(C_2, C_1)$ becomes true if $C_1 \sqsubseteq C_2$.

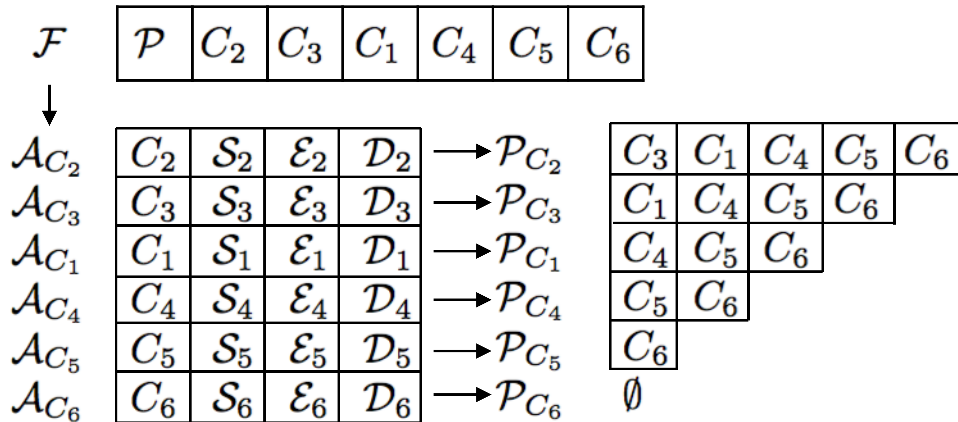


FIGURE 5.1: Initialization of half-matrix

Definition 5.1 A 2-DiHM (2-Dimensional Half Matrix) is a tuple $\langle \mathcal{P}, \mathcal{N}_{\mathcal{O}} \rangle$, where $\mathcal{N}_{\mathcal{O}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ contains all satisfiable concepts of an ontology $\mathcal{O}$ (or TBox), $\mathcal{P}$ is a finite set of potential possible subsumees of all concepts in $\mathcal{N}_{\mathcal{O}}$, where n is the total number of concepts.

Definition 5.2 A quadruple $\mathcal{A}_{\mathcal{C}_i} = \langle \mathcal{C}_i, \mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i \rangle$ contains known information for every satisfiable $\mathcal{C}_i \in N_{\mathcal{O}}$, where $\mathcal{S}_i$ contains $\mathcal{C}_i$'s direct subsumees, $\mathcal{E}_i$ $\mathcal{C}_i$'s equivalent concepts including itself, and $\mathcal{D}_i$ $\mathcal{C}_i$'s disjoint concepts.

For every pair $\{\mathcal{C}_i, \mathcal{C}_j\} \in N_{\mathcal{O}}$ with $i \neq j$, all the concepts in both $\mathcal{X}_{\mathcal{E}_i}$ ($\mathcal{X}_{\mathcal{D}_i}$) and $\mathcal{X}_{\mathcal{E}_j}$ ($\mathcal{X}_{\mathcal{D}_j}$) are disjoint.

Definition 5.3 A set $\mathcal{R}_{\mathcal{O}}$ is defined as $\mathcal{R}_{\mathcal{O}} = \bigcup_{\mathcal{C}_i \in N_{\mathcal{O}}} \{\mathcal{P}_{\mathcal{C}_i}\}$, which reflects all possible sets $\mathcal{P}_{\mathcal{C}_i}$ where $\mathcal{P}_{\mathcal{C}_i} \neq \emptyset$.

Example 5.1 In an ontology $\mathcal{O}$, there are six satisfiable concepts in $\mathcal{N}_{\mathcal{O}}$, that has $\mathcal{F} = \{\mathcal{P}, C_1, C_2, C_3, C_4, C_5, C_6\}$ and $C_2 \gtrdot C_3 \gtrdot C_1 \gtrdot C_4 \gtrdot C_5 \gtrdot C_6$. For every concept $C_i \in \mathcal{N}_{\mathcal{O}}$, $\mathcal{F}$ and $\mathcal{A}_{C_i}$ are created as shown in Figure 5.1.

For all the satisfiable concepts in $N_{\mathcal{O}}$, using a half-matrix structure represents all possible relations with other concepts inferred or tested by a black-box reasoner, e.g.,

SUBS?(C_2, C_1) becomes true if $C_1 \sqsubseteq C_2$. Assume there are four satisfiable concepts $\{C_1, C_2, C_6, C_3\} \in N_{\mathcal{O}}$ and the tested relations among these concepts are $\mathcal{O} \models \{C_1 \sqsubseteq C_2,\ C_2 \not\sqsubseteq C_1,\ C_6 \not\sqsubseteq C_3,\ C_3 \sqsubseteq \mathcal{C}_6\}$. Accordingly the changes to $\mathcal{P}$ and $\mathcal{A}$ result in the following sets:

$$\begin{array}{ll}
\mathcal{A}_{C_2} \rightarrow \mathcal{S}_2 = \langle \emptyset, \{C_1\} \rangle & \mathcal{P}_{C_2} = \{C_3, \not{C_1}, C_4, C_5, C_6\} \\
\mathcal{A}_{C_3} \rightarrow \mathcal{S}_3 = \langle \{C_6\}, \emptyset \rangle & \mathcal{P}_{C_3} = \{C_1, C_4, C_5, \not{C_6}\} \\
\mathcal{A}_{C_1} \rightarrow \mathcal{S}_1 = \langle \{C_2\}, \emptyset \rangle & \mathcal{P}_{C_1} = \{C_4, C_5, C_6\} \\
\mathcal{A}_{C_4} \rightarrow \mathcal{S}_4 = \langle \emptyset, \emptyset \rangle & \mathcal{P}_{C_4} = \{C_5, C_6\} \\
\mathcal{A}_{C_5} \rightarrow \mathcal{S}_5 = \langle \emptyset, \emptyset \rangle & \mathcal{P}_{C_5} = \{C_6\} \\
\mathcal{A}_{C_6} \rightarrow \mathcal{S}_6 = \langle \emptyset, \{C_3\} \rangle & \mathcal{P}_{C_6} = \emptyset
\end{array}$$

Therefore, we obtain two subsumption testing results for every pair of concepts, which guarantee the completeness and less memory used when make the changes in $\mathcal{P}$ and $\mathcal{A}$ for every pair of tests until all concepts $\mathcal{C}_i \in \mathcal{N}_{\mathcal{O}}$ have been tested.

Algorithm 12: PARALLELCLASSIFICATION

input : Ontology $\mathcal{O}$, Black-box Reasoner R

1 CREATEHALF-MATRIXSTRUCTURE
2 $T \leftarrow$ CREATETHREADPOOL
3 **while** GETALLAXIOMS **do**
4     $A \leftarrow$ AXIOMDIVISION
5   **for** *each axiom* $A_i \in A$ **do**
6   **if** SCHEDULEWORK(T) **then**
7   PRECOMPUTING(A_i)
8 **while** GETREMAININGPOSSIBLESET **do**
9     $G \leftarrow$ GROUPDIVISION
10   **for** *each group* $G_i \in G$ **do**
11   **if** SCHEDULEWORK(T) **then**
12   CLASSIFICATIONSUBTEST(G_i, T)
13 COMPUTEONTOLOGYTAXONOMY
14 **return**

15 **procedure** CREATEHALF-MATRIXSTRUCTURE
16     $N_{\mathcal{O}} \leftarrow$ GETALLSATCONCEPTS
17   **for** *each concept* $C_i \in N_{\mathcal{O}}$ **do**
18   CREATE $\mathcal{A}_{\mathcal{C}_i} = \langle \mathcal{C}_i, \mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i \rangle$ and $\mathcal{P}_{\mathcal{C}_i}$
19   DEFINEORDER$(N_{\mathcal{O}})$

20 **procedure** DEFINEORDER($N_{\mathcal{O}}$)
21   **return** $\mathcal{C}_a > \mathcal{C}_b > ... > \mathcal{C}_c > \mathcal{C}_d ... > \mathcal{C}_i > \mathcal{C}_j$

22 **procedure** SCHEDULEWORK(T)
23     $T_i \leftarrow$ GETAVAILABLETHREAD(T)
24   STARTBLACK-BOXREASONER(T_i)
25   **return** T_i

5.2.2 Maintaining Sets

Maintaining (Direct) Subsumers or Subsumees

Given $\mathcal{S}_i, \mathcal{S}_j$ with $i \neq j$ Definition 5.4 states rules to maintain (direct) subsumers or subsumees. Upon termination of processing, the sets $\mathcal{H}_{i\uparrow}$ ($\mathcal{H}_{i\downarrow}$) contain the direct subsumers (subsumees) to construct the complete subsumption hierarchy.

Definition 5.4 Given $\mathcal{A}_{\mathcal{C}_i} = \langle \mathcal{C}_i, \mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i \rangle$ of $\mathcal{C}_i \in \mathcal{N}_\mathcal{O}$, $\mathcal{S}_i$ is defined as $\mathcal{S}_i = \langle \mathcal{H}_{i\uparrow}, \mathcal{H}_{i\downarrow} \rangle$, where $\mathcal{H}_{i\uparrow}$ contains current direct subsumers of $\mathcal{C}_i$ and $\mathcal{H}_{i\downarrow}$ subsumees of $\mathcal{C}_i$.

The related sets of concepts $\mathcal{C}_i$ and $\mathcal{C}_j$ ($\mathcal{C}_i > \mathcal{C}_j$) have $\mathcal{H}_{i\uparrow} = \emptyset, \mathcal{H}_{i\downarrow} = \emptyset$, $\mathcal{H}_{j\uparrow} = \emptyset, \mathcal{H}_{j\downarrow} = \emptyset$, $\mathcal{H}_{k\uparrow} = \{\mathcal{C}_i\}, \mathcal{H}_{k\downarrow} = \{\mathcal{C}_j\}$, $\mathcal{H}_{l\uparrow} = \{\mathcal{C}_i\}$ and $\mathcal{H}_{l\downarrow} = \{\mathcal{C}_j\}$, then

- *if* $\mathcal{H}_{i\uparrow} \cap \mathcal{H}_{j\downarrow} = \emptyset, \mathcal{C}_i \sqsubseteq \mathcal{C}_j$ and $\mathcal{C}_j \not\sqsubseteq \mathcal{C}_i$, *then* $\mathcal{H}_{j\downarrow} = \{\mathcal{C}_i\}, \mathcal{H}_{i\uparrow} = \{\mathcal{C}_j\}$.

- *if* $\mathcal{H}_{i\uparrow} \cap \mathcal{H}_{j\downarrow} \neq \emptyset, \mathcal{C}_i \sqsubseteq \mathcal{C}_j$ and $\mathcal{C}_j \not\sqsubseteq \mathcal{C}_i$, *then* $\mathcal{H}_{i\uparrow} = \emptyset$ and $\mathcal{H}_{j\downarrow} = \emptyset$, since $\mathcal{C}_j$ ($\mathcal{C}_i$) is not a direct subsumer (subsumee) of $\mathcal{C}_i$ ($\mathcal{C}_j$).

- *if* $\mathcal{H}_{i\uparrow} \cap \mathcal{H}_{j\uparrow} \neq \emptyset, \mathcal{C}_i \sqsubseteq \mathcal{C}_j$ and $\mathcal{C}_j \not\sqsubseteq \mathcal{C}_i$, *then* $\mathcal{H}_{i\uparrow} - (\mathcal{H}_{i\uparrow} \cap \mathcal{H}_{j\uparrow}) = \mathcal{H}_{i\uparrow}$, since $\forall \mathcal{C}$ ($\mathcal{C} \in \mathcal{H}_{i\uparrow} \cap \mathcal{H}_{j\uparrow}$) are indirect subsumers of $\mathcal{C}_i$.

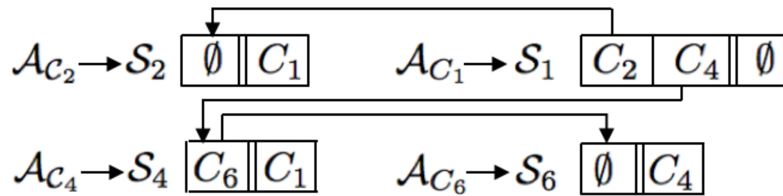
- *if* $\mathcal{H}_{i\downarrow} \cap \mathcal{H}_{j\downarrow} \neq \emptyset, \mathcal{C}_i \sqsubseteq \mathcal{C}_j$ and $\mathcal{C}_j \not\sqsubseteq \mathcal{C}_i$, *then* $\mathcal{H}_{j\downarrow} - (\mathcal{H}_{i\downarrow} \cap \mathcal{H}_{j\downarrow}) = \mathcal{H}_{j\downarrow}$, since $\forall \mathcal{C}$ ($\mathcal{C} \in \mathcal{H}_{i\downarrow} \cap \mathcal{H}_{j\downarrow}$) are indirect subsumees of $\mathcal{C}_j$.

For every pair of concepts C_i and C_j, the subsumer (subsumee) set $\mathcal{H}_{i\uparrow}$ ($\mathcal{H}_{i\downarrow}$) is linked to the subsumer (subsumee) set $\mathcal{H}_{j\uparrow}$ ($\mathcal{H}_{j\downarrow}$) when the subsumption relation $C_i \sqsubseteq C_j$ ($C_j \sqsubseteq C_i$) is found. The related operations for equivalent sets are shown in UPDATE-SUBCLASS(C_i, C_j) of Algorithm 13.

Algorithm 13: UPDATESUBCLASS(C_i, C_j)

```
1  procedure UPDATESUBCLASS(C_i, C_j)
2    S_i = {∅,∅} S_j = {∅,∅}
3    if SUBS?(C_i, C_j) then
4      H_{i↑} = {C_j}, H_{j↓} = {C_i}
5      if H_{j↑} then
6          DELETE C ∈ H_{j↑} in P_{C_i}
7      if SUBS?(C_j, C_i) then
8          H_{j↑} = {C_i}, H_{i↓} = {C_j}
9          if H_{i↑} then
10             DELETE C ∈ H_{i↑} in P_{C_j}
11         UPDATEEQUIVALENT(C_i, C_j)
12     DELETE C_j in P_{C_i}
13   DELETE C_i in P_{C_j}
```

Assume $C_1 \sqsubseteq C_4$, $C_1 \sqsubseteq C_2$ and $C_4 \sqsubseteq C_6$ are known, then $\mathcal{S}_1 = \langle\{C_2, C_4\}, \emptyset\rangle$, $\mathcal{S}_2 = \langle\emptyset, C_1\rangle$, $\mathcal{S}_4 = \langle\{C_6\}, \{C_1\}\rangle$ and $\mathcal{S}_6 = \langle\emptyset, \{C_4\}\rangle$, which is illustrated below by using solid arrows to indicate the subsumer relations among these concepts and the two sets $\mathcal{H}_{i\uparrow}$ and $\mathcal{H}_{i\downarrow}$ separated by double solid lines.



Maintaining Equivalence and Disjointness

Definition 5.5 A $\mathcal{B}$-type $\langle\mathcal{E}, \mathcal{D}\rangle$ tuple is defined as $\mathcal{B} = \langle\mathcal{C}, \mathcal{X}\rangle$, where $\mathcal{C}$ is a set to contain a concept $\mathcal{C}_i$ for mapping with other concepts in set $\mathcal{X}$, which have equivalent

or disjoint relationship with $\mathcal{C}$ and the sequence is defined in DEFINEORDER (see line 20-21 in Algorithm 12).

- *if* $\mathcal{X}_i \cap \mathcal{X}_j \neq \emptyset, \mathcal{C}_{\mathcal{B}_i} = \mathcal{C}_i$ and $\mathcal{C}_{\mathcal{B}_j} = \mathcal{C}_j$, *then* update $\mathcal{C}_{\mathcal{B}_j} = \mathcal{C}_i$ as index, $\mathcal{X}_j \cup \mathcal{X}_i = \mathcal{X}_i$ and $\mathcal{X}_j = \emptyset$.

- *if* $\mathcal{X}_i \cap \mathcal{X}_j = \emptyset$, $\mathcal{C}_{\mathcal{B}_i} = \mathcal{C}_i$ and $\mathcal{X}_i \neq \emptyset$ but $\mathcal{X}_j = \emptyset$, $\mathcal{C}_{\mathcal{B}_j} = \mathcal{C}_k$ for $\mathcal{C}_k$ in $\mathcal{A}_{\mathcal{C}_k}$ $(k \neq i, k \neq j)$, *then* use $\mathcal{C}_{\mathcal{B}_j}$ as index to find $\mathcal{B}_k$ for $\mathcal{C}_{\mathcal{B}_k} = \mathcal{C}_k$, update both $\mathcal{C}_{\mathcal{B}_j} = \mathcal{C}_i$, $\mathcal{C}_{\mathcal{B}_k} = \mathcal{C}_i$, $\mathcal{X}_k \cup \mathcal{X}_i = \mathcal{X}_i$, and $\mathcal{X}_k = \emptyset$.

- *if* $\mathcal{X}_i = \emptyset, \mathcal{C}_{\mathcal{B}_i} = \mathcal{C}_k$ that $\mathcal{C}_k$ in $\mathcal{A}_{\mathcal{C}_k}$ and $\mathcal{X}_j = \emptyset, \mathcal{C}_{\mathcal{B}_j} = \mathcal{C}_k$ that $\mathcal{C}_l$ in $\mathcal{A}_{\mathcal{C}_l}$, *then* use $\mathcal{C}_{\mathcal{B}_i}, \mathcal{C}_{\mathcal{B}_j}$ as index to find $\mathcal{B}_k$ and $\mathcal{B}_l$ $(l \neq i, l \neq j, l \neq k)$, i.e. $\mathcal{C}_{\mathcal{B}_k} = \mathcal{C}_k, \mathcal{C}_{\mathcal{B}_k} = \mathcal{C}_l$, update $\mathcal{C}_{\mathcal{B}_i} = \mathcal{C}_k, \mathcal{C}_{\mathcal{B}_l} = \mathcal{C}_k$, $\mathcal{X}_l \cup \mathcal{X}_k = \mathcal{X}_k$ and $\mathcal{X}_l = \emptyset$.

It holds for every pair $\mathcal{B}_i, \mathcal{B}_j$ in $\mathcal{B}$ with $i \neq j$ and $\mathcal{B}_i = \langle \mathcal{C}_{\mathcal{B}_i}, \mathcal{X}_i \rangle$ and $\mathcal{B}_j = \langle \mathcal{C}_{\mathcal{B}_j}, \mathcal{X}_j \rangle$, that $\mathcal{X}_i \sqcap \mathcal{X}_j \sqsubseteq \bot$.

Definition 5.6 Given $\mathcal{A}_{\mathcal{C}_i} = \langle \mathcal{C}_i, \mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i \rangle$ of $\mathcal{C}_i \in \mathcal{N}_{\mathcal{O}}$, $\mathcal{E}_i$ is defined as $\mathcal{B}$-type $\mathcal{E}_i = \langle \mathcal{C}_{\mathcal{E}_i}, \mathcal{X}_{\mathcal{E}_i} \rangle$, where $\mathcal{C}_{\mathcal{E}_i}$ is the current mapping concept with other equivalent concepts in $\mathcal{X}_{\mathcal{E}_i}$ and $\mathcal{C}_{\mathcal{E}_i} = \mathcal{C}_i$, $\mathcal{X}_{\mathcal{E}_i} = \{\mathcal{C}_i\}$ initially.

There are concepts $C_i, C_j, C_l, C_k, C_m, C_n$ and $\mathcal{C}_m \equiv \mathcal{C}_n$, then

- *if* $\mathcal{C}_{\mathcal{E}_m} = \mathcal{C}_m$ and $\mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_n$, *then* $\mathcal{P}_{\mathcal{C}_m} = \mathcal{P}_{\mathcal{C}_m} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{E}_n}\}$, $\mathcal{X}_{\mathcal{E}_m} = \mathcal{X}_{\mathcal{E}_m} \cup \mathcal{X}_{\mathcal{E}_n}$, $\mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_m$, both $\mathcal{X}_{\mathcal{E}_n} = \emptyset$ and $\mathcal{P}_{\mathcal{C}_n} = \emptyset$.

- *if* $\mathcal{C}_{\mathcal{E}_m} = \mathcal{C}_m$ and $\mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_l$, *then* $\mathcal{P}_{\mathcal{C}_l} = \mathcal{P}_{\mathcal{C}_l} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{E}_m}\}$, $\mathcal{X}_{\mathcal{E}_l} = \mathcal{X}_{\mathcal{E}_l} \cup \mathcal{X}_{\mathcal{E}_m}$, $\mathcal{C}_{\mathcal{E}_m} = \mathcal{C}_l$, $\mathcal{X}_{\mathcal{E}_m} = \emptyset$, $\mathcal{X}_{\mathcal{E}_n} = \emptyset$, $\mathcal{P}_{\mathcal{C}_m} = \emptyset$ and $\mathcal{P}_{\mathcal{C}_n} = \emptyset$.

- *if* $\mathcal{C}_{\mathcal{E}_m} = \mathcal{C}_k$ and $\mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_l$, *then* $\mathcal{P}_{\mathcal{C}_k} = \mathcal{P}_{\mathcal{C}_k} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{E}_l}\}$, $\mathcal{X}_{\mathcal{E}_k} = \mathcal{X}_{\mathcal{E}_k} \cup \mathcal{X}_{\mathcal{E}_l}$, $\mathcal{C}_{\mathcal{E}_l} = \mathcal{C}_k, \mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_k$, $\mathcal{X}_{\mathcal{E}_l} = \emptyset$, $\mathcal{X}_{\mathcal{E}_m} = \emptyset$, $\mathcal{X}_{\mathcal{E}_n} = \emptyset$, $\mathcal{P}_{\mathcal{C}_l} = \emptyset$, $\mathcal{P}_{\mathcal{C}_m} = \emptyset$ and $\mathcal{P}_{\mathcal{C}_n} = \emptyset$.

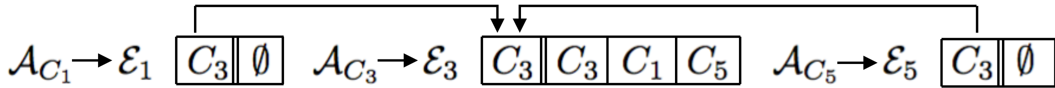
- *if* $\mathcal{C}_m \sqsubseteq \mathcal{C}_j, \mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_m$ and $\mathcal{X}_{\mathcal{E}_m} \neq \emptyset$, *then* $\mathcal{H}_{j\uparrow} = \mathcal{H}_{j\uparrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{E}_m}\}$.
- *if* $\mathcal{C}_i \sqsubseteq \mathcal{C}_m, \mathcal{C}_{\mathcal{E}_n} = \mathcal{C}_m$ and $\mathcal{X}_{\mathcal{E}_m} \neq \emptyset$, *then* $\mathcal{H}_{i\downarrow} = \mathcal{H}_{i\downarrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{E}_m}\}$.

Therefore, if the related sets of concepts $\mathcal{C}_i$ and $\mathcal{C}_j$ ($\mathcal{C}_i \gtrdot \mathcal{C}_j$) have $\mathcal{E}_i = \langle \mathcal{C}_i, \{\mathcal{C}_i\}\rangle$, $\mathcal{E}_j = \langle \mathcal{C}_j, \{\mathcal{C}_j\}\rangle$, $\mathcal{C}_i \sqsubseteq \mathcal{C}_j$ and $\mathcal{C}_j \sqsubseteq \mathcal{C}_i$, i.e. $\mathcal{C}_i \equiv \mathcal{C}_j$, then $\mathcal{E}_i = \langle \mathcal{C}_i, \{\mathcal{C}_i, \mathcal{C}_j\}\rangle$ and $\mathcal{E}_j = \langle \mathcal{C}_i, \emptyset\rangle$. The related operations for equivalent sets are shown in UPDATEEQUIVALENT(C_i, C_j) of Algorithm 14.

Algorithm 14: UPDATEEQUIVALENT(C_i, C_j)

1 **procedure** UPDATEEQUIVALENT(C_i, C_j)
2 $\quad \mathcal{C}_a \leftarrow$MAPPINGEQUIVALENT($C_i$)
3 $\quad \mathcal{C}_b \leftarrow$MAPPINGEQUIVALENT($C_j$)
4 $\quad$ CHECKDEFINEDORDER($\mathcal{C}_a, \mathcal{C}_b$)
5 $\quad$ **if** $\mathcal{E}_b \backslash (\mathcal{E}_b \cap \mathcal{E}_a) \neq \emptyset$ **then**
6 $\quad\quad$ DELETE $\mathcal{E}_b \backslash (\mathcal{E}_b \cap \mathcal{E}_a)$ in $\mathcal{P}_{\mathcal{C}_a}$
7 $\quad \mathcal{P}_{\mathcal{C}_a} = \emptyset,\ \mathcal{E}_a = \mathcal{E}_a \cup \mathcal{E}_b,\ \mathcal{E}_b = \emptyset$

Assuming $C_1 \equiv C_3$ and $C_3 \equiv C_5$, the changes of $\mathcal{E}_i$ become $\mathcal{E}_1 = \langle \mathcal{C}_3, \emptyset\rangle$, $\mathcal{E}_3 = \langle C_3, \{C_3, C_1, C_5\}\rangle$ and $\mathcal{E}_5 = \langle C_3, \emptyset\rangle$ shown below (equivalent mapping directions linked with solid arrows and the sets $\mathcal{C}$ and $\mathcal{X}$ separated by double solid lines).



Definition 5.7 Given $\mathcal{A}_{\mathcal{C}_i} = \langle \mathcal{C}_i, \mathcal{S}_i, \mathcal{E}_i, \mathcal{D}_i\rangle$ of $\mathcal{C}_i \in \mathcal{N}_{\mathcal{O}}$, $\mathcal{D}_i$ defined as $\mathcal{B}$-type $\mathcal{D}_i = \langle \mathcal{C}_{\mathcal{D}_i}, \mathcal{X}_{\mathcal{D}_i}\rangle$, where $\mathcal{C}_{\mathcal{D}_i}$ contains the current mapping with other disjoint concepts in $\mathcal{X}_{\mathcal{D}_i}$ and $\mathcal{C}_{\mathcal{D}_i} = \mathcal{C}_i$, $\mathcal{X}_{\mathcal{D}_i} = \emptyset$ initially.

There are concepts $C_i, C_j, C_l, C_k, C_m, C_n$ and $\mathcal{C}_k \sqcap \mathcal{C}_l \sqsubseteq \bot$, then

- *if* $\mathcal{C}_{\mathcal{D}_k} = \mathcal{C}_k, \mathcal{C}_{\mathcal{D}_l} = \mathcal{C}_l$, *then* $\mathcal{P}_{\mathcal{C}_l} = \mathcal{P}_{\mathcal{C}_l} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_k}\}, \mathcal{P}_{\mathcal{C}_k} = \mathcal{P}_{\mathcal{C}_k} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_l}\}, \mathcal{X}_{\mathcal{D}_k} = \mathcal{X}_{\mathcal{D}_k} \cup \{\mathcal{C}_l\}$ and $\mathcal{X}_{\mathcal{D}_l} = \mathcal{X}_{\mathcal{D}_l} \cup \{\mathcal{C}_k\}$.

- *if* $\mathcal{C}_{\mathcal{D}_k} = \mathcal{C}_i, \mathcal{C}_{\mathcal{D}_l} = \mathcal{C}_l$, *then* $\mathcal{P}_{\mathcal{C}_l} = \mathcal{P}_{\mathcal{C}_l} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_i}\}, \mathcal{P}_{\mathcal{C}_i} = \mathcal{P}_{\mathcal{C}_i} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_l}\}, \mathcal{X}_{\mathcal{D}_l} = \mathcal{X}_{\mathcal{D}_l} \cup \{\mathcal{C}_i\}$ and $\mathcal{X}_{\mathcal{D}_i} = \mathcal{X}_{\mathcal{D}_i} \cup \{\mathcal{C}_l\}$.

- *if* $\mathcal{C}_{\mathcal{D}_k} = \mathcal{C}_i, \mathcal{C}_{\mathcal{D}_l} = \mathcal{C}_j$, *then* $\mathcal{P}_{\mathcal{C}_j} = \mathcal{P}_{\mathcal{C}_j} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_i}\}, \mathcal{P}_{\mathcal{C}_i} = \mathcal{P}_{\mathcal{C}_i} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_j}\}, \mathcal{X}_{\mathcal{D}_j} = \mathcal{X}_{\mathcal{D}_j} \cup \{\mathcal{C}_i\}, \mathcal{X}_{\mathcal{D}_i} = \mathcal{X}_{\mathcal{D}_i} \cup \mathcal{C}_j$.

- *if* $\mathcal{C}_l \sqsubseteq \mathcal{C}_j$, $\mathcal{C}_{\mathcal{D}_k} = \mathcal{C}_k$ and $\mathcal{X}_{\mathcal{D}_k} \neq \emptyset$, *then* $\mathcal{H}_{j\uparrow} = \mathcal{H}_{j\uparrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_k}\}$ and $\mathcal{H}_{j\downarrow} = \mathcal{H}_{j\downarrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_k}\}$.

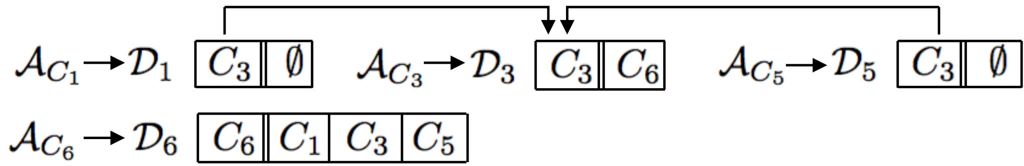
- *if* $\mathcal{C}_i \sqsubseteq \mathcal{C}_k$, $\mathcal{C}_{\mathcal{D}_l} = \mathcal{C}_l$ and $\mathcal{X}_{\mathcal{D}_l} \neq \emptyset$, *then* $\mathcal{H}_{i\uparrow} = \mathcal{H}_{i\uparrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_l}\}$ and $\mathcal{H}_{i\downarrow} = \mathcal{H}_{i\downarrow} - \{\forall \mathcal{C}_x | \mathcal{C}_x \in \mathcal{X}_{\mathcal{D}_l}\}$.

Therefore, if the related sets of concepts $\mathcal{C}_i$ and $\mathcal{C}_j$ ($\mathcal{C}_i > \mathcal{C}_j$) have $\mathcal{D}_i = \langle \mathcal{C}_i, \emptyset \rangle$, $\mathcal{D}_j = \langle \mathcal{C}_j, \emptyset \rangle$, $\mathcal{C}_i \sqcap \mathcal{C}_j \sqsubseteq \perp$, then $\mathcal{D}_i = \langle \mathcal{C}_i, \{\mathcal{C}_j\} \rangle$ and $\mathcal{D}_j = \langle \mathcal{C}_j, \{\mathcal{C}_i\} \rangle$. The related operations for equivalent sets are shown in UPDATEDISJOINT(C_i, C_j) of Algorithm 15.

Algorithm 15: UPDATEDISJOINT(C_i, C_j)

1 **procedure** UPDATEDISJOINT(C_i, C_j)
2 $\mathcal{C}_c \leftarrow$MAPPINGDISJOINT(C_i)
3 $\mathcal{C}_d \leftarrow$MAPPINGDISJOINT(C_j)
4 CHECKDEFINEDORDER($\mathcal{C}_c, \mathcal{C}_d$)
5 **if** $(\mathcal{D}_c \cup \mathcal{D}_d) \backslash (\mathcal{D}_c \cap \mathcal{D}_d) \neq \emptyset$ **then**
6 DELETE
7 $\mathcal{D}_d \backslash (\mathcal{D}_d \cap \mathcal{D}_c)$ in $\mathcal{P}_C$ for $C \in \mathcal{S}_c$,
8 $\mathcal{D}_c \backslash (\mathcal{D}_d \cap \mathcal{D}_c)$ in $\mathcal{P}_C$ for $C \in \mathcal{S}_d$
9 DELETE C_d in $\mathcal{P}_{C_c}$, C_c in $\mathcal{P}_{C_d}$
10 $\mathcal{D}_c = \mathcal{D}_c \cup \{\mathcal{C}_d\}$, $\mathcal{D}_d = \mathcal{D}_d \cup \{\mathcal{C}_c\}$

Assuming disjoint concepts $C_1 \sqcap C_6 \sqsubseteq \bot$, then $C_3 \sqcap C_6 \sqsubseteq \bot$ and $C_5 \sqcap C_6 \sqsubseteq \bot$ are inferred by $C_1 \equiv C_3$ and $C_3 \equiv C_5$. The changes of $\mathcal{D}_i$ become $\mathcal{D}_1 = \langle C_3, \emptyset \rangle$, $\mathcal{D}_3 = \langle C_3, \{C_6\} \rangle$, $\mathcal{D}_5 = \langle C_3, \emptyset \rangle$ and $\mathcal{D}_6 = \langle C_6, \{C_1, C_3, C_5\} \rangle$ shown below (disjoint mapping directions linked with solid arrows and the sets $\mathcal{C}$ and $\mathcal{X}$ separated by double solid lines).



Example 5.2 Following Example 5.1, the following relations among the concepts are hold: $\mathcal{O} \models \{C_1 \equiv C_5, C_3 \equiv C_4, C_6 \sqsubseteq C_2, C_3 \sqsubseteq C_6, C_2 \sqcap C_5 \sqsubseteq \bot\}$.

Since $C_3 \sqsubseteq C_6, C_6 \sqsubseteq C_2$ and $C_3 \equiv C_4$, we can infer that $C_3 \sqsubseteq C_2$ and $\{C_3, C_4\} \in \mathcal{S}_6$. With reference to UPDATESUBSUMEE (see Algorithm 13), the changes to the subsumee sets are $\mathcal{S}_2 = \{C_6\}$, $\mathcal{S}_3 = \emptyset$, $\mathcal{S}_1 = \emptyset$, $\mathcal{S}_4 = \emptyset$, $\mathcal{S}_5 = \emptyset$ and $\mathcal{S}_6 = \{C_3, C_4\}$. According to UPDATEEQUIVALENT (see Algorithm 14), $C_1 \equiv C_5$, $C_3 \equiv C_4$ and $C_2 \gtrdot C_3 \gtrdot C_1 \gtrdot C_4 \gtrdot C_5$ are known, so the changes to the equivalent sets are $\mathcal{E}_2 = \{C_2\}$, $\mathcal{E}_3 = \{C_3, C_4\}$, $\mathcal{E}_1 = \{C_1, C_5\}$, $\mathcal{E}_4 = \emptyset$, $\mathcal{E}_5 = \emptyset$ and $\mathcal{E}_6 = \{C_6\}$. Because of $C_2 \sqcap C_5 \sqsubseteq \bot$ and $C_1 \equiv C_5$, $C_2 \sqcap C_1 \sqsubseteq \bot$ is inferred. Since C_3, C_4 and C_6 are subsumees of C_2, therefore, both C_1 and C_5 are disjoint with C_3, C_4 and C_6. Based on UPDATEDISJOINT (see Algorithm 15), the changes to the disjoint sets are $\mathcal{D}_2 = \{C_1, C_5\}$, $\mathcal{D}_3 = \emptyset$, $\mathcal{D}_1 = \{C_2, C_6, C_3, C_4\}$, $D_4 = \emptyset$, $D_5 = \emptyset$ and $\mathcal{D}_6 = \emptyset$. The complete changes are shown in Figure 5.2, which indicates that all the subsumption relations among the six concepts have been found and there are no more subsumption tests required, which results in $\mathcal{R}_{\mathcal{O}} = \emptyset$.

Let us reconsider the changes of set $\mathcal{A}_{C_1}$ in Example 4.5 (see Figure 4.9 and 4.10). First, since $C_1 \equiv C_5$ is known, C_5 is added to $\mathcal{E}_1$. Second, we know that C_2, C_6, C_3 and

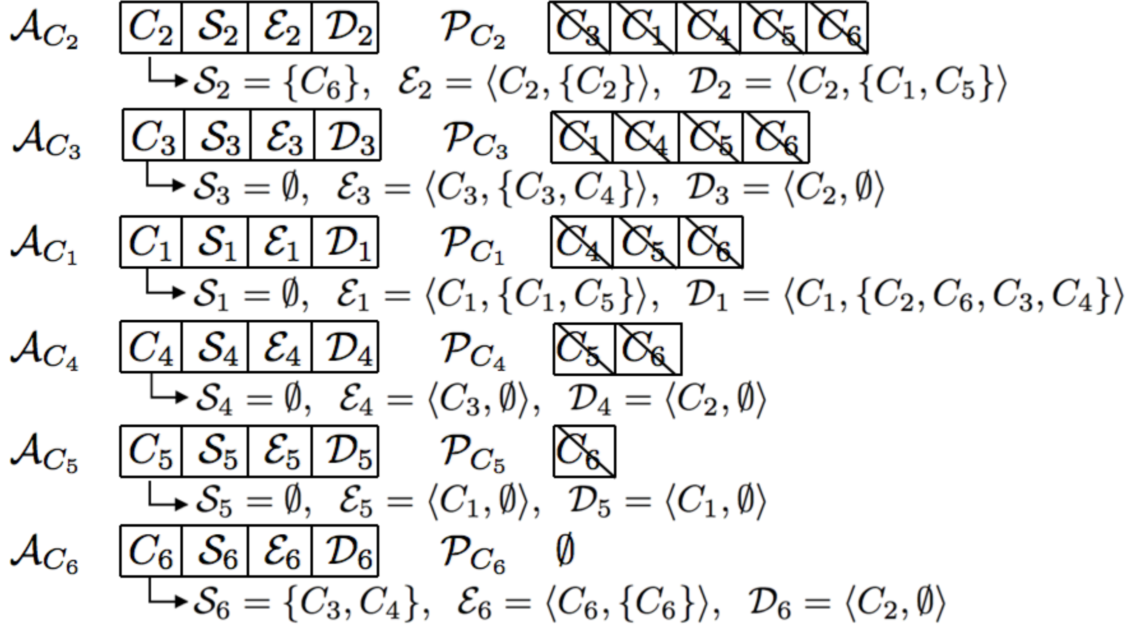


FIGURE 5.2: Complete changes of $\mathcal{F}$ after applying rules

C_4 are disjoint with C_1 when $C_2 \sqcap C_1 = \bot$, $C_6 \sqsubseteq C_2$, $C_3 \equiv C_4$ and $C_3 \sqsubseteq C_6$ reasoning using different threads. Therefore, when C_2, C_6, C_3 and C_4 are added to $\mathcal{D}_1$, there might exist potential conflicts when modifying $\mathcal{D}_1$. In order to guarantee exclusive write access for all the processors at the same time without being interrupted, an atomic operation is used to ensure currently running process cannot be interrupted. Therefore, the problem mentioned above can be solved as shown in Figure 5.3. The solid arrows indicate the sequence of making the changes in $\mathcal{A}_1$ and the potential conflicts mentioned above can be resolved by having steps 3 and 5 to make sure the unique access of $\mathcal{D}_1$ in the half-martix.

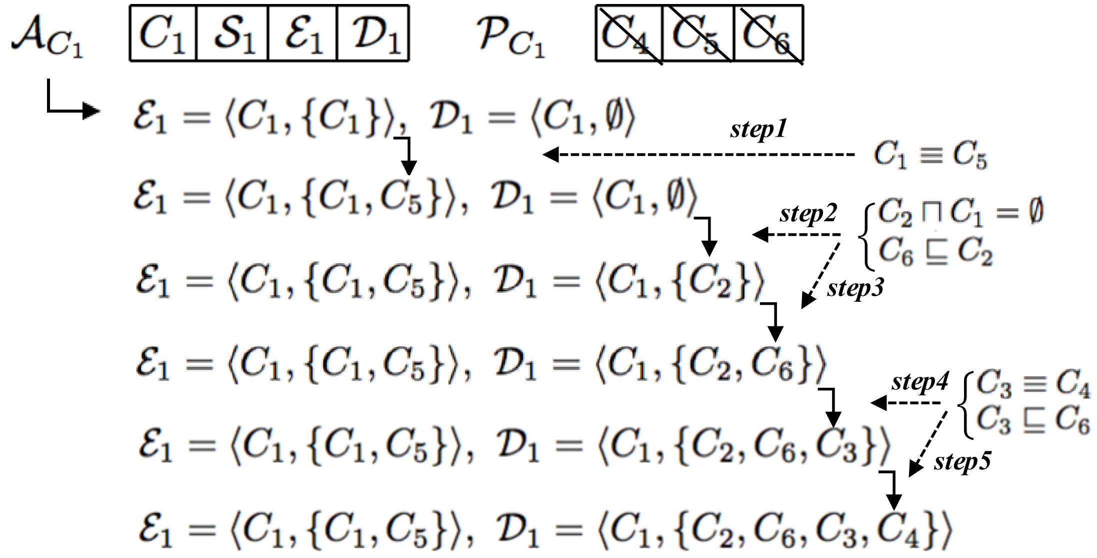


FIGURE 5.3: Using atomic operations to solve conflicts in $\mathcal{A}_1$

5.3 Improved Ontology Classification

5.3.1 Precomputing Phase

In the precomputing part, OWL API [26] is applied to retrieve all declared axioms of an ontology $\mathcal{O}$, and a pool of axioms is created to store these axioms. Whenever a subsumption can be directly derived from an axiom, e.g., $A \sqsubseteq B$, if the converse subsumption is unknown, it is tested using the chosen black-box reasoner, e.g., SUBS?(A, B). Because of different kinds of potential relations among concepts, currently three kinds of relations are covered: subClass ($\mathcal{S}$), equivalence ($\mathcal{E}$) and disjointness ($\mathcal{D}$) axioms (see Algorithm 16).

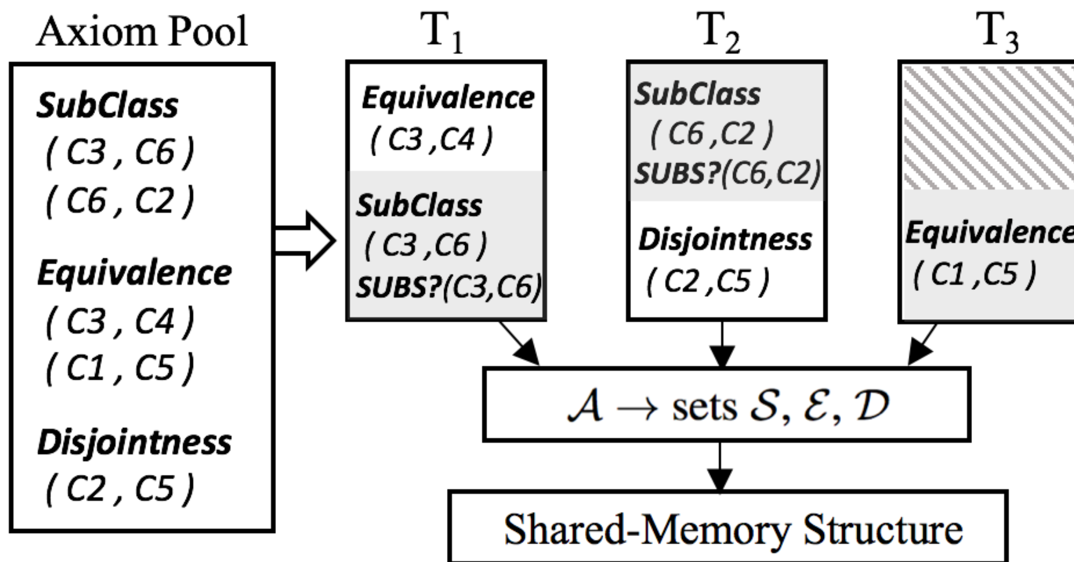


FIGURE 5.4: Parallel precomputing phase

Algorithm 16: PRECOMPUTING(A_i)

```
1 for each pair {C_i, C_j} ∈ A_i do
2     if SubClass (C_i, C_j) then
3         UPDATESUBSUMEE(C_i, C_j)
4     else if Equivalence (C_i, C_j) then
5         UPDATEEQUIVALENCE(C_i, C_j)
6     else if Disjointness (C_i, C_j) then
7         UPDATEDISJOINTNESS(C_i, C_j)
```

In Example 5.2, the OWL input can be interpreted as shown in Figure 5.4, which has an axiom pool containing the identified axioms and three threads (T_1, T_2, T_3) to analyze the results. From the results shown in Figure 5.2, all the possible sets are empty, which means all the possible relations among the six satisfiable concepts have been tested or inferred and the results are recorded in sets $\mathcal{S}, \mathcal{E}, \mathcal{D}$ of $\mathcal{A}$ respectively.

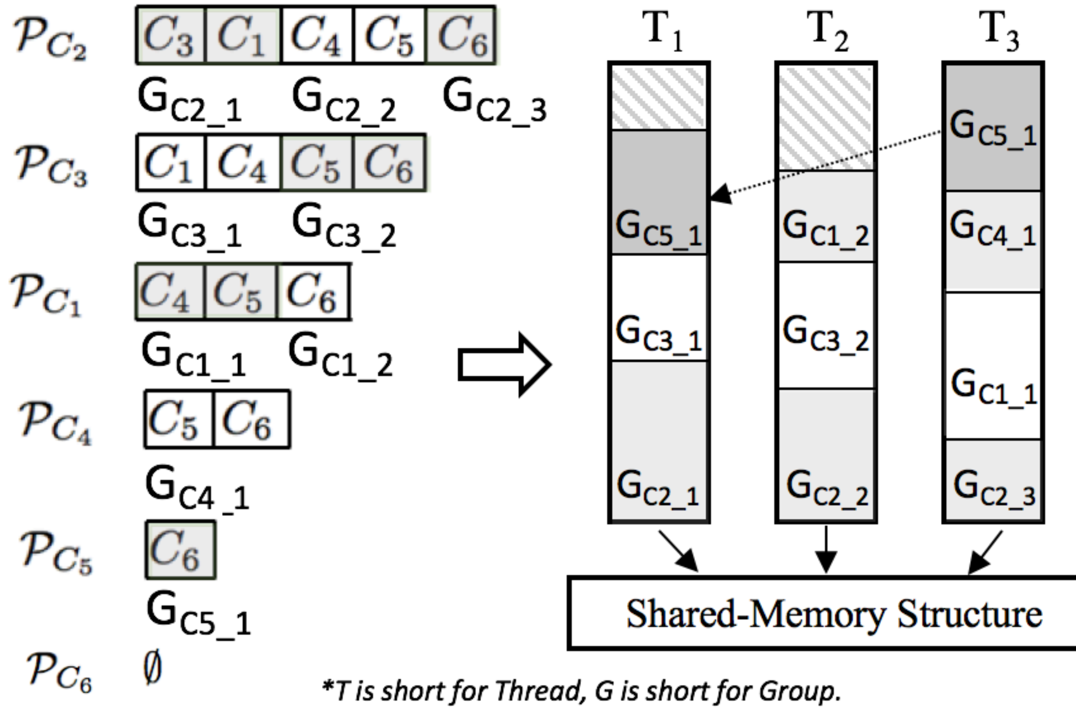


FIGURE 5.5: *Work-Stealing* strategy applied between T_1 and T_3

5.3.2 Classification Phase

However, it is possible that $\mathcal{R}_O$ is not empty after the precomputing phase, then the classification phase is processed to finish the classification and guarantee the completeness of this method. Because of the differences of every subsumption test performed by black-box reasoners, it it important to ensure concurrency and avoid longer waiting time for the rest of concepts especially when the tests are taking longer than estimated. A work-stealing strategy is applied to schedule different threads dynamically and improve load balancing among threads to speed up the classification process.

Work-Stealing Strategy

First, find all the remaining possible $\mathcal{P}_{C_i} \in \mathcal{R}_O$ with $\mathcal{P}_{C_i} \neq \emptyset$. Second, separate $\mathcal{R}_O$ into smaller subgroups G_i and put them into a queue Q_i. The sizes of groups depend on the remaining size of $\mathcal{P}$ and the number of processor n currently available. Third,

Algorithm 17: CLASSIFICATIONSUBTEST(G_i, T)

```
1  ENQUEUE(Q_i, G_i)
2  for each pair {C_i, C_j} ∈ Q_i do
3      UPDATESUBSUMEE(C_i, C_j)
4      DEQUEUE(Q_i, {C_i, C_j})
5  if ¬ISEMPTY(Q_i) then
6      STEALWORK(T, Q_i)
7  procedure STEALWORK(T, Q_j)
8    if SCHEDULEWORK(T) then
9        for each pair {C_m, C_n} ∈ Q_j do
10           UPDATESUBSUMEE(C_m, C_n)
11           DEQUEUE(Q_j, {C_m, C_n})
12   if ¬ISEMPTY(Q_j) then
13       STEALWORK(T, Q_j)
```

if there is an idle thread available during the classification process, a new group from the queue will be given to that thread dynamically until all the subgroups have been classified and $\mathcal{R}_O$ is empty (see Algorithm 17).

Example 5.3 Using the six concepts generated in Example 5.2, all the concepts in $\mathcal{P}$ are divided into subgroups G_i and put into a queue Q. As shown in Figure 5.5, all the generated subgroups are indicated by the colors grey or white to separate them. Suppose there are three threads (T_1, T_2, T_3) available, then three queues will be generated for each thread, e.g., $Q_1 = \{G_{C_{2_1}}, G_{C_{3_1}}\}$, $Q_2 = \{G_{C_{2_2}}, G_{C_{3_2}}, G_{C_{1_2}}\}$, $Q_3 = \{G_{C_{2_3}}, G_{C_{1_1}}, G_{C_{4_1}}, G_{C_{5_1}}\}$. During the classification, when all tasks of Q_1 assigned in T_1 have been finished, a task $G_{C_{5_1}}$ (see Figure 5.5 in darker grey) needs to be done by T_3, which is currently working on $G_{C_{4_1}}$. Therefore, the task $G_{C_{5_1}}$ will be

stolen from T_3 and reallocated to T_1. Accordingly, all the updated information will be recorded in $\mathcal{A}$ as well.

After classification, all the relevant information of each concept $\mathcal{C}_i$ is recorded in $\mathcal{A}_{C_i}$. According to $\mathcal{A}_{C_i}$, the whole taxonomy of ontology $\mathcal{O}$ is computed.

Theorem 1 (Soundness) Let $\mathcal{A}_i, \mathcal{P}_{C_i}$ be a complete set for concept C_i and $\mathcal{A}_j, \mathcal{P}_{C_j}$ for concept C_j. If the subsumption relations between a pair $\{C_i, C_j\}$ are correctly inferred by sound black-box reasoners, e.g., $\textsc{subs?}(C_i, C_j)$ and $\textsc{subs?}(C_j, C_i)$, or the algorithms of maintaining sets (see Algorithm 13, 14, 15), which do not conclude a wrong subsumption relation between two concepts, then this parallel method is sound for $\mathcal{O}$.

Theorem 2 (Completeness) For all the satisfiable concepts $\mathcal{C}_i \in N_{\mathcal{O}}$, both $\mathcal{A}_i$ and $\mathcal{P}_{C_i}$ of $\mathcal{C}_i$ are created completely. All the possible relations among concepts are recorded in $\mathcal{P}$. A subsumption test for each pair $\{C_i, C_j\}$ $(i \neq j)$ is performed either by a complete black-box reasoner via $\textsc{subs?}(\mathcal{C}_i, \mathcal{C}_j)$ and $\textsc{subs?}(\mathcal{C}_j, \mathcal{C}_i)$ or by maintaining sets (see Algorithm 13, 14, 15). Therefore, the set $\mathcal{R}_{\mathcal{O}}$ is empty if and only if all the possible relations in the sets $\mathcal{P}_{C_i}$ have been derived.

5.4 Summary

In this part, an improved parallel OWL ontology classification method is presented based one the first version described in Chapter 4. Compared to the previous version, this method applies two phases - precomputing and classification to achieve a better performance when compete with the black-box reasoners. Furthermore, due to the improvements of our atomic data structure, more potential subsumption relations can be reduced without testing by the black-box reasoners.

6 Evaluation

In this part, we use the first version parallel method (see Chapter 4) to evaluate the results of ontology scale, complexity and load balancing with increasing number of threads. Secondly, the improved version (see Chapter 5) is used to compete with original black-box reasoners in precomputing and complete classification process.

6.1 First Evaluation

6.1.1 Benchmarks

The parallel classification architecture is implemented as a Java shared-memory program using HermiT 1.3.8 as OWL reasoner plug-in. The experiments are performed on a HP DL580 Scientific Linux[1] SMP server with four 15-core processors (Gen8 Intel Xeon E7-4890v2 2.8GHz) and 1 TB RAM.

For the first evaluation of the classification architecture, a set of 9 real-world ontologies are selected from the ORE 2015 [45] repository that contain up to 13,000 concepts and 33,000 axioms to test scalability and 6 ontologies from the ORE 2014 [44] repository that contain up to 7,000 axioms and 967 qualified cardinality restrictions (QCRs) (see Section 2.2.1), which are used to constrain the number of values of a particular property and type and are considered to be an important parameter in testing the

[1] GNU/Linux Version 2.6.32-642.15.1.el6.x86_64

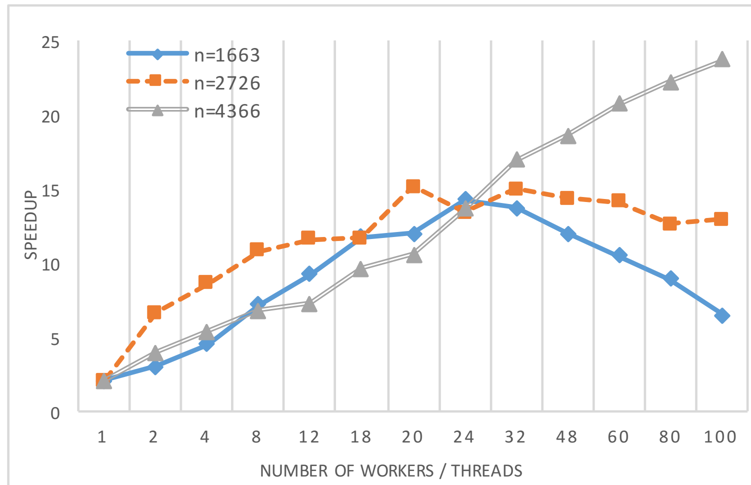
complexity factors of this approach. Their metrics are shown in Tables 6.1+6.4 (see Section 2.2.1 about naming DLs). For benchmarking we ensured exclusive access to the server in order to avoid that other jobs affect the elapsed time of the tests. For tests with a smaller set of threads we ran several jobs in parallel but jobs exceeding 60 threads were run exclusively.

TABLE 6.1: Metrics of tested OWL ontologies

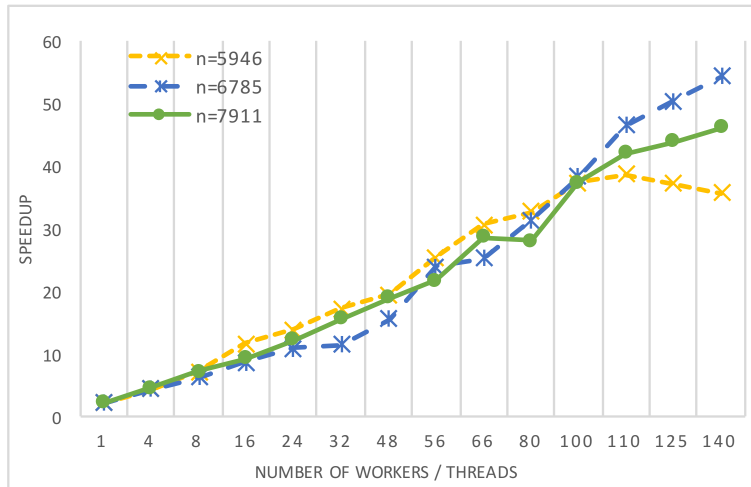
Ontology	Concept	Axiom	SubClass	Expressivity
WBbt.obo	6785	19138	12347	$\mathcal{EL}$
EHDA#EHDA	8341	33367	8339	$\mathcal{EL}$
obo.PREVIOUS	1663	4099	1377	$\mathcal{ELH}+$
actpathway.obo	7911	25314	17402	$\mathcal{EL}$
EHDAA2	2726	16818	13458	$\mathcal{ELH}+$
lanogaster.obo	10925	16567	5641	$\mathcal{EL}$
MIRO#MIRO	4366	21274	4454	$\mathcal{EL}+$
CLEMAPA	5946	16864	10916	$\mathcal{EL}$
EMAP#EMAP	13735	27467	13732	$\mathcal{EL}$

6.1.2 Ontology Scale

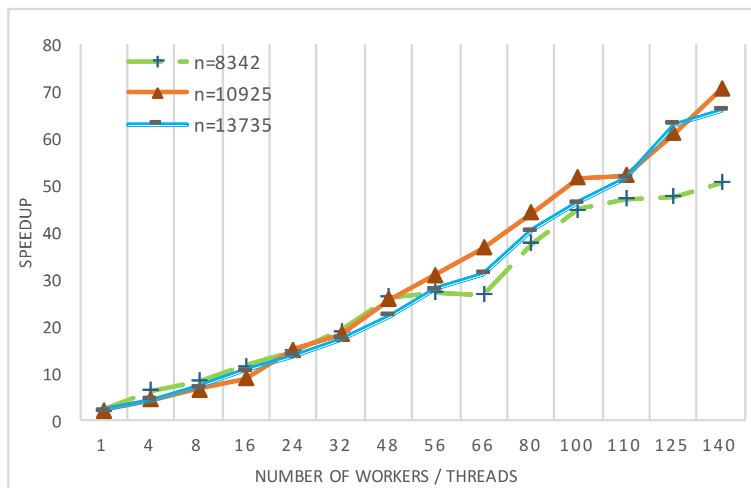
In order to assess the scalability of the architecture, a series of experiments are conducted where the number of workers/threads available for classification varied between 1 (sequential case) to 140. Due to the limitations of the test environment we restricted the maximum number of threads to 140. We computed the speedup as the ratio of the runtime (sum of runtimes of all threads) divided by the elapsed time. Each



(a) $n \in \{1663, 2726, 4366\}$



(b) $n \in \{5946, 6785, 7911\}$



(c) $n \in \{8341, 10925, 13735\}$

FIGURE 6.1: Speedup factors for ontologies from Table 6.1 with an increasing number of concepts (n = number of concepts)

individual experiment was repeated three times and the resulting average was used to determine its runtime and elapsed time. The 9 ontologies can be roughly divided into three groups of similar sizes measured by their number of contained concepts (n).

Figure 6.1(a) shows a set of smaller ontologies. For the two smallest ontologies the peak speedup is reached with 20-32 workers. A higher number of workers indicates a performance degradation that is due to the current partitioning scheme where the size of the partition allocated to of each worker is roughly $\frac{n}{w}$ (n is the number of concepts in an ontology and w the number of workers). When the partition size becomes too small, overhead affects the performance adversely.

Figure 6.1(b) shows medium-sized and Figure 6.1(c) large ontologies. With the exception of the smallest ontology in Figure 6.1(b) both figures show a similar speedup increase. This is due to bigger partition sizes and reduced overhead. The peak is currently reached with 140 workers. It is necessary to make partition sizes reasonably big.

6.1.3 Ontology Complexity

There are other factors that can affect the experiments such as the complexity of an ontology and the efficiency of HermiT, the selected plug-in reasoner, which is also implemented in Java. For most of the used ontologies we observed that the runtimes of individual subsumption tests performed by HermiT are rather uniform but for ontologies with a higher expressivity it is well known that just a few subsumption tests may require a significant amount of the total runtime. Furthermore, the plug-in reasoner might be more or less efficient depending on the expressivity of the test ontologies.

In order to test the performance of this architecture for complex ontologies, we

TABLE 6.2: Metrics of the used OWL ontologies with QCRs

Ontology	Concept	Axiom	Sub	QCR	Expressivity
nskisimple_functional	1737	4775	2234	43	$\mathcal{SRIQ}(\mathcal{D})$
ncitations_functional	2332	7304	2786	47	$\mathcal{SROIQ}(\mathcal{D})$
ddiv2_functional	1469	4080	1832	48	$\mathcal{SRIQ}(\mathcal{D})$
rnao_functional	731	2884	1235	446	$\mathcal{SRIQ}$
jectOWLDL2_functional	482	1093	325	425	$\mathcal{ALN}$
bridg.biomedical_domain	320	6347	295	967	$\mathcal{SROIN}(\mathcal{D})$

used the same experimental environment and selected six smaller real-world ontologies with a logic of high expressivity as shown in Table 6.4, which lists for each ontology its expressivity, number of concepts, axioms, subclasses, equivalent classes, disjoint classes, QCRs, existential and universal restrictions.

Since the maximum number of concepts for these ontologies is 2332, experiments are conducted where the number of available workers range from 1 to 100. We computed the speedup as the ratio of runtime divided by elapsed time. Each experiment was repeated three times and the resulting runtime and elapsed time averages were used to calculate the speedup. We roughly divided the six ontologies into two groups based on their number of QCRs and speedup. Moreover, in order to better understand how the performance of the plug-in reasoner and thus the runtimes of individual subsumption tests affect the results, we collected for the six tested ontologies statistics about subsumption test runtimes (in milliseconds) such as minimum, maximum, average, median, and deviation (see Table 6.3).

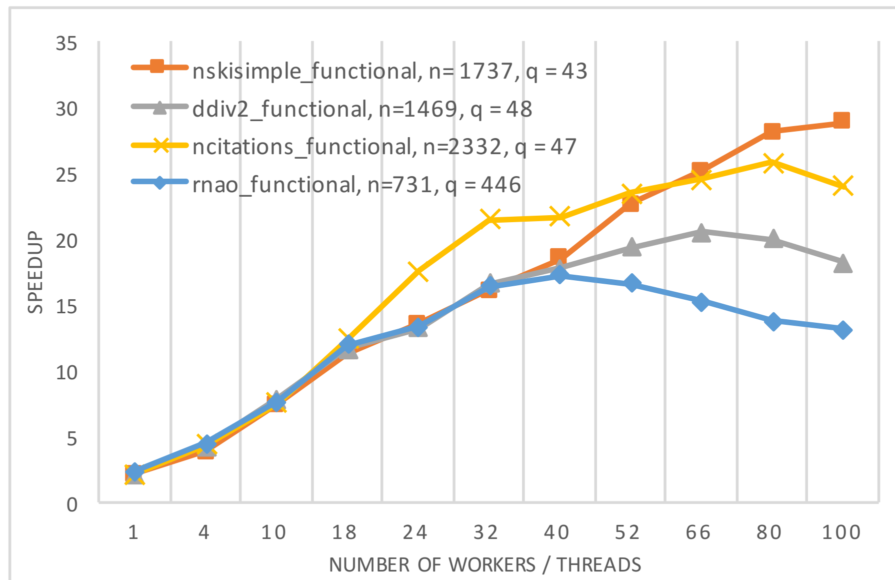
In Figure 6.2(a), the number of QCRs in the first group ranges between 40-446. Since we try to select reasonable partition sizes, we used up to 100 threads to compute the speedup factors for all four ontologies. As the number of threads is increased, a

TABLE 6.3: Time metrics using 10 workers (in milliseconds)
(Ave = Average, Med = Median, Dev = Deviation)

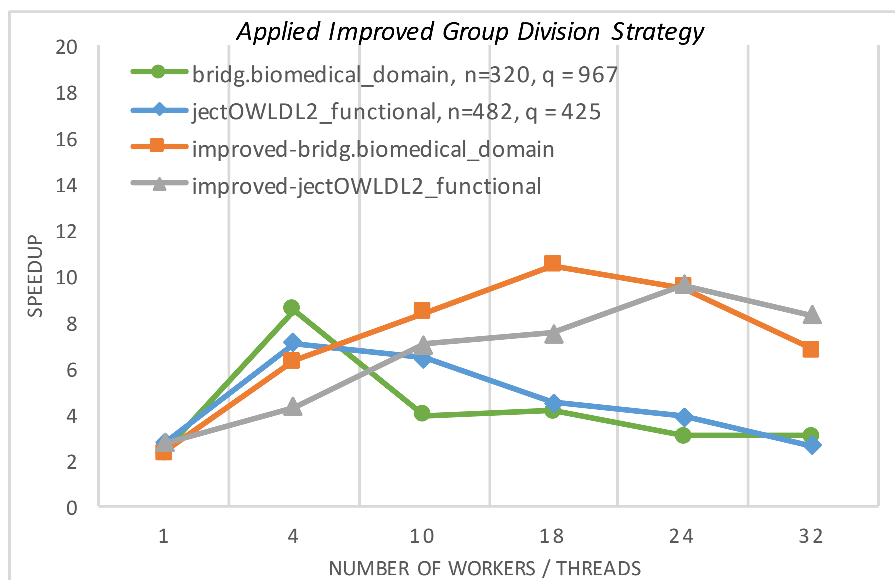
Ontology	Min	Max	Ave	Med	Dev
nskisimple_functional	18.23	440.29	39.86	195.29	108.34
ncitations_functional	17.54	711.58	27.95	176.14	251.25
ddiv2_functional	10.66	300.89	27.35	19.59	44.64
rnao_functional	17.18	206.96	66.47	92.49	144.15
jectOWLDL2_functional	0.004	231.56	0.033	0.09	36.90
bridg.biomedical_domain	0.004	357.62	0.036	0.95	48.71

better speedup is observed and the maximum is reached with 60-100 threads except for the one with q = 446, which has small-sized concepts and reached its maximum speedup around 40 threads. Table 6.3 shows that average runtimes are similar but deviation is several orders of magnitude higher than the average, which does not affect the experimental results significantly. From these results we also can see that if the subsumption tests become more complex, i.e., they take longer, the optimized method can also achieve a good speedup for ontologies of smaller sizes. The speedup is even better compared to a similarly sized ontology such as *obo.PREVIOUS* (see Table 6.1 and Figure 6.1(a)).

In Figure 6.2(b), the number of QCRs is reaching 425 (n=482) and 967 (n=320), which indicates the difficulty of ontology classification. Due to the complexity and limitations of HermiT, these two ontologies show the best performance for four workers and afterwards the speedup factor remains around 4. As we observed, these ontologies include some difficult QCRs, which cause several subsumption tests to take much longer than others (as indicated in Table 6.3 by a very high deviation), therefore their speedup does not always increase.



(a) $q \in \{43, 47, 48, 446\}$



(b) $q \in \{425, 967\}$

FIGURE 6.2: Speedup factors for ontologies with QCRs from Table 6.4 (q = number of QCRs)

In order to improve the performance for these complex ontologies, we used the *Fork/Join framework* already mentioned in the improved group division strategy (see Section 4.4.2) to reschedule tasks which require a significantly longer runtime for subsumption tests. The old and improved results are shown in Figure 6.2(b). Because of

dividing bigger tasks into smaller ones by using *work stealing*, compared with the old results a continuously increasing speedup factor can be achieved until the maximum with around 20 workers has been reached.

As expected, in general the results show that this method has a speedup linear to the number of threads. Due to the new group division strategy, the scheduling of idle threads achieves a better load balancing.

6.1.4 Load Balancing

From the experiments we observed that the first (random division) phase (with randomly created groups of similar average size) exhibits a better load balancing than the second (group division) phase. However, the classification process can only terminate once the second phase has been completed. To get a better understanding of the performance for both the random division and the group division phase, we used a ratio representing the decrease of the number of possible subsumers in each phase.

Definition 6.1 *InitialPossible* is defined as the initial number of possible subsumers for an ontology and *RemainingPossible* is the number of possible subsumers after completing each division cycle. Therefore, the possible ratio is defined in (6.1) as follows.

$$Possible = \frac{InitialPossible - RemainingPossible}{InitialPossible} \tag{6.1}$$

We chose the ontology *ncitations_functional.owl* from Table 6.4 with 2332 concepts and used 10 workers. We decided on ten random division cycles and one group division cycle to determine the load balance factors. We also recorded the runtime for each phase and calculated the runtime ratio as the accumulated cycle runtimes divided by the total runtime. The result is shown in Figure 6.3.

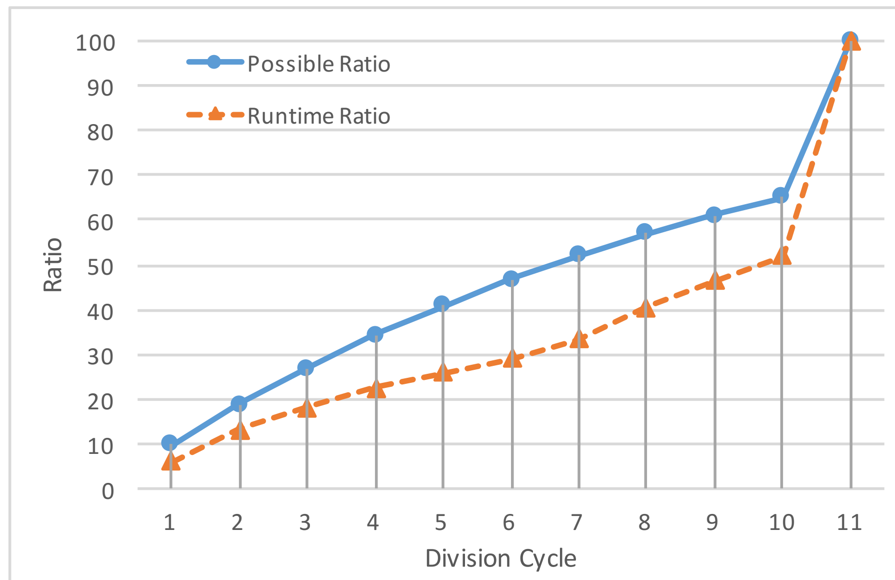


FIGURE 6.3: Division cycle result of *ncitations_functional.owl* (concepts = 2332, threads = 10, random division cycle = 10, group division cycle = 1)

Since we implemented two parallel classification phases in the methods and the random division phase applied a completely random division strategy to minimize *possible* on a large scale. As expected, the random division strategy (cycles 1-10) increased the value *Possible* up to 60, i.e., the number of possible subsumees was reduced by 60%, before the group division strategy was applied. The runtime ratio is almost at the same level as the possible ratio (see Figure 6.3). However, from the test results we noticed that with an increasing number of threads, the ratio factor not necessarily increases too, especially if the number of threads is more than 60. We are still working on finding a better load balancing between the two phases which can both shorten the runtime and reduce the number of possible subsumees as quickly as possible. Therefore, the ratio factor affecting load balancing of the two parallel phases can be expected to be improved when much larger ontologies are tested.

6.1.5 Summary

The first version parallel method is tested using the sequential OWL reasoner HermiT. For some difficult ontologies this method can outperform the stand-alone version of HermiT. However, due to processor and reasoner restrictions, not all ontologies could be tested on the current platform within a reasonable amount of time. From the test results which are evaluated by a set of real-world ontologies, the experiments demonstrate a very good scalability resulting in a speedup that is linear to the number of available cores.

6.2 Improved Parallel Classification

In order to achieve a better performance compared to the first version and compete with the black-box reasoners, we also tested the improved parallel framework, which is implemented with shared-memory half-matrix structure of *Atomic*, which is a toolkit of variable in Java Concurrent package. The chosen black-box OWL reasoner is used for deciding concept satisfiability and subsumption, and integrates Java Concurrency Framework, which supports generating more than one thread to maximize CPU utilization. We performed the evaluation by exclusively using a HP DL580 Scientific Linux SMP server with four 15-core processors and a total of 1 TB RAM (each processor has 256 GB of shared RAM). Due to the limitations of the current experimental environment, we used at most 120 threads for every experiment. In order to better compare the performance of the system with other popular OWL reasoners [20], the test ontologies were selected from the ORE [44] repository to evaluate the performance of this parallel approach. They vary by the number of axioms, concepts, and for precomputing by the number of subclass, equivalence and disjointness axioms. Considering the implementations of different reasoners and their Java compatibility, currently

we successfully applied this parallel reasoning framework by using two OWL reasoners as black-box reasoners: (i) Hermit 1.3.8 [22] is an OWL reasoner fully supporting OWL datatypes; and (ii) JFact 5.0.3 is a Java port of FaCT++ [55], a tableau based OWL reasoner. For reasons of compatibility and performance, we mainly focus on the comparison and evaluation with Hermit.

6.2.1 Benchmarks

We tested the parallel framework using two reasoners (Hermit and JFact) individually. The wall clock time is recorded for the precomputing phase and whole classification process separately. The current experimental environment allows us to use up to 120 threads. The actual number of threads depends on the ontology's size and reasoning difficulty. All the experiments were repeated five times and the resulting average is used to determine the wall clock time and speedup factors. Table 6.4 shows the characteristics of 10 selected ontologies including the number of axioms, named concepts, subClass, equivalent and disjoint axioms. In the precomputing phase, an axiom pool is created to contain all the axioms eligible for precomputing. Axiom preprocessing is parallelized using the maximum number of threads allowed. In order to test the performance of precomputing, we tested both the sequential and parallel cases using different numbers of threads (20, 60, 100, 120) with Hermit. The results (wall clock time (WCT) in seconds) are shown in Table 6.5. The best result is indicated for each ontology in bold.

In order to better assess the impact of the overhead due to parallelization and other potential factors such as the efficiency of the selected black-box reasoner, we also recorded time statistics of subsumption tests performed by the black-box reasoner, such as deviation, maximum, minimum, median, and average time. Table 6.6 reports various time metrics and data for 11 different ontologies over the whole classification

TABLE 6.4: Metrics of tested ontologies for precomputing
(Equi = Equivalence Axioms, Disjoint = Disjointness Axioms)

Ontology	Axiom	Concept	SubClass	Equi	Disjoint
microbial.type	13,584	4,636	7,255	935	31
MSC_classes	13,584	5,559	8,220	930	382
CURRENT	26,374	6,595	17,180	2,297	218
natural.product	169,498	9,463	12,370	0	56,192
vertebrate	94,564	18,092	71,579	4,428	0
pr_simple	149,568	59,006	89,854	0	693
attributes	221,783	62,035	141,224	18,029	137
CLASSIFIED	169,155	83,036	55,046	30,363	693
behavior	354,825	99,360	241,046	14,013	62
havioredit	354,971	99,399	241,140	14,026	62

process. Table 6.7 presents the wall clock time of the system with (Para) and without using work-stealing (PW), the times of Hermit, and the speedup factors, which are calculated by dividing the wall clock time of PW by Para and Hermit by Para. In addition, the results of a larger set of ontologies compare with Hermit is shown in Section 6.2.5. The best results are all indicated in bold.

6.2.2 Precomputing Phase

Table 6.5 shows that the precomputing time could be significantly improved due to parallelization by using up to 120 threads (bold font indicates the best time). The ontologies *microbial.type* and *CURRENT* could be processed about 600 times faster than in the sequential case when 100 or 120 threads are applied. The ontology *vertebrate*

TABLE 6.5: Precomputing Results using Hermit
(timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds, T = number of threads, >1,000 = more than 1000 times)

Ontology	PreComputing WCT					Speedup
	T = 1	**T = 20**	**T = 60**	**T = 100**	**T = 120**	**Factor**
microbial.type	304	93.9	17.5	**0.5**	3.2	608
MSC_classes	156	61.9	9.7	**1.46**	2.83	107
CURRENT	391	182	10.9	8.7	**0.74**	528
natural.product	67.9	23.6	8.91	3.48	**2.16**	31.5
vertebrate	TO	TO	TO	TO	TO	>1,000
pr_simple	TO	464	147	105	**38.2**	>1,000
attributes	TO	860	630	218	**120**	>1,000
CLASSIFIED	TO	TO	TO	883	**79.2**	>1,000
behavior	TO	TO	972	729	**303**	>1,000
havioredit	TO	TO	TO	TO	TO	-

timed out even for 120 threads due to the black-box reasoner that is already used in the precomputing phase. The next four bigger ontologies timed out if only one thread is used but could be processed with an increasing number of threads and lead to a speedup of more than 1,000 compared to the sequential case. The biggest ontology *havioredit* still timed out for 120 threads. Due to the use of parallelization and a atomic half-martix shared-memory structure together with the maximum number of available threads, a better performance is achieved by updating accumulative information and reducing the total number of subsumption tests, which results in a decreased wall clock time in the precomputing phase compared to the sequential case.

6.2.3 Improved Classification

TABLE 6.6: Time Metrics of tested OWL ontologies
(timeout (TO) = 1,000 seconds, Dev = Deviation, Med = Median, Ave = Average)

Ontology	Axiom	Concept	Subsumption Test Statistics				
			Dev	**Max**	**Min**	**Med**	**Ave**
mfoem.emotion	2,389	902	0.26	1.92	0.03	0.22	0.12
nskisimple	4,775	1,737	0.07	0.42	0.0001	0.23	0.03
geolOceanic	6,573	2,324	0.21	1.01	0.017	0.55	0.07
stateEnergy	10,270	3,018	1.94	9.78	0.07	1.12	0.25
aksmetrics	11,134	3,889	0.73	1.24	0.005	0.34	0.21
microbial.type	13,584	4,636	2.15	13.6	0.03	3.38	1.13
MSC_classes	15,092	5,559	-	TO	-	-	-
CURRENT	26,374	6,595	10.9	32.8	0.01	6.34	2.72
compatibility	21,720	7,929	4.37	9.38	0.005	3.72	0.98
natural.product	169,498	9,463	12.5	87.2	0.02	15.8	6.93
havioredit	354,971	99,399	-	TO	-	-	-

Table 6.6 + 6.7 indicate two important factors affecting the performance of this system: the partition size and the efficiency of subsumption tests. A reasonable partition size for each thread can reduce the overhead of waiting or updating information in the atomic half-matrix structure, e.g., *mfoem.emotion* and *nskisimple* are more than 10 times faster than Hermit when 80-100 threads are applied since each thread has a reasonable partition size and less overhead according to the deviation that is closer to the average time. When the size of ontologies increases, such as for *geolOceanic, stateEnergy*, and *aksmetrics*, a better performance is achieved with 100-120 threads because of reasonable partition sizes and uniformity of subsumption tests, which result in smaller

TABLE 6.7: Improved classification results using Hermit
(timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds)

Ontology	Axiom	Concept	WCT		Speedup
			Parallel	Hermit	Factor
mfoem.emotion	2,389	902	**2.7**	42.1	15.6
nskisimple	4,775	1,737	**2.9**	29.3	10.1
geolOceanic	6,573	2,324	**1.4**	12.1	8.6
stateEnergy	10,270	3,018	**12.3**	72.9	5.9
aksmetrics	11,134	3,889	**3.3**	13.6	4.2
microbial.type	13,584	4,636	**26.7**	308	11.5
MSC_classes	15,092	5,559	TO	TO	-
CURRENT	26,374	6,595	**112**	452	4.0
compatibility	21,720	7,929	**20.2**	22.5	1.1
natural.product	169,498	9,463	98.7	**11.21**	0.1
havioredit	354,971	99,399	TO	TO	-

differences between deviation and average time.

In order to better assess the impact of black-box reasoners on the framework, we computed more statistics on subsumption tests that are also shown in Table 6.6. The statistics lists 9.78s as maximum time for *stateEnergy*. Thus, the performance of the framework cannot be below that maximum time. *MSC_classes* times out for Hermit and this framework. The individual subsumption tests are performed by the black-box reasoners, and its effectiveness also constrains the performance of this framework, i.e., if a single subsumption test times out as indicated for *MSC_classes*, then the system times out also due to the black-box reasoner. For the ontology *microbial.type*, many subsumptions can be derived during parallel precomputing, which results in a speedup

factor of almost 600 (see Table 6.4). Moreover, if tested sequentially, this ontology requires some difficult tests which take more time than the maximum of 13.6s (parallel testing). Due to parallel processing and fast accumulation and synchronous updating of concept relations in the atomic structure, the system can avoid these difficult tests, which makes this framework more than 10 times faster than the black-box reasoner.

When the size of ontologies increases even more, such as for *CURRENT*, where many subsumptions can be derived during parallel precomputing, we achieved a speedup of 4 with 120 threads. For *compatibility*, which has about 8,000 concepts, the performance of this approach is below but close to the black-box reasoner, because some subsumption tests could be avoided by black-box reasoner optimizations but were required for the framework in order to guarantee completeness. However, for the second last ontology *natural.product* this system cannot compete with Hermit because the maximum subsumption time is very high and it seems that black-box reasoner optimizations can avoid this test that are inaccessible to the system due to the black-box approach. The last ontology *havioredit*, which is the biggest one we chose, times out for all the reasoners and this framework. Each thread is overloaded by the number of concepts to classify, which results in more overhead in the whole classification process and, thus, causes a timeout for *havioredit*, even though the precomputing phase becomes faster.

Overall, this optimized parallel framework achieves a better performance than Hermit when enough threads are available to ensure reasonable partitions for different ontology sizes, especially if the number of concepts is less than 10,000.

6.2.4 Load Banlancing

In order to get a better understanding the performance of the parallel framework when work-stealing is applied, the improved version of this method is applied work-stealing

TABLE 6.8: Improved classification results using Hermit (timeout (TO) = 1,000 seconds, WCT = wall clock time in seconds, P(ara) = Parallel, W = without work stealing)

Ontology	Axiom	Concept	WCT		Speedup
			Para	**PW**	**Factor**
mfoem.emotion	2,389	902	**2.7**	25.3	9.4
nskisimple	4,775	1,737	**2.9**	28.1	9.7
geolOceanic	6,573	2,324	**1.4**	19.2	13.7
stateEnergy	10,270	3,018	**12.3**	201	16.3
aksmetrics	11,134	3,889	**3.3**	46.5	14.1
microbial.type	13,584	4,636	**26.7**	512	19.2
MSC_classes	15,092	5,559	TO	TO	-
CURRENT	26,374	6,595	**112**	783	6.9
compatibility	21,720	7,929	**20.2**	240	11.9
natural.product	169,498	9,463	**98.7**	352	3.6
havioredit	354,971	99,399	TO	TO	-

strategy to make a balance of group distribution, which is caused by each subsumption tested by different black-box reasoners. Therefore, Table 6.8 shows the wall clock time of the parallel framework without applying work stealing (PW) and in the second last column the speedup factors defined by $\frac{PW}{Para}$. From the results, the best performances have a factor of 19.2 and 16.3 for the ontology *microbial.type* and *stateEnergy* respectively, which have a high maximum time compared to the wall clock time of Para. Most of the improved speedup factors are in the range of 9-15, which show the improvements when the work-stealing strategy is applied in this approach.

6.2.5 Comparison with DL Reasoners

There are two larger sets of ontologies tested including the ones used in the Section 6.2. The results present the wall clock time of this framework compared with Hermit or JFact. The wall clock time for sequential and parallel precomputing is listed in Table 6.9 + 6.10. Ontologies are sorted by their number of concepts.

6.2.6 Summary

Using the improved parallel classification framework, the results demonstrate the performance of this parallel framework against the selected black-box reasoner by classifying a great variety of ontologies. The results show that the wall clock time of the parallel framework has better results when the ontologies can be classified by black-box reasoner. However, since the efficiency of the subsumption tests is constrained by the black-box reasoner and due to the limitation of the current experimental environment (a total of 60 hyper-threading cores supporting 120-150 threads), the results outperform the black-box reasoner when the size of ontologies are less than 10,000 concepts in most cases.

TABLE 6.9: Time Metrics of tested OWL ontologies using parallel framework with Hermit (timeout (TO) = 1000 seconds) (Sequ = Sequential, Para = Parallel)

#	Ontology	Concept	Precomputing		Classification		Speedup
			Sequ	Para	Para	Hermit	Factor
1	SocialUnits	156	84.3	**0.86**	**16.1**	353	21.9
2	00021	156	105	**0.91**	**15.4**	260	16.9
3	rnao.owl	240	1.16	**0.29**	**3.06**	109	35.9
4	tionmodule	256	523	**1.12**	**640**	909	1.42
5	genetic	386	671	**8.80**	**31.2**	530	17.0
6	WM30	415	TO	**0.74**	TO	**798**	-
7	ainability	824	6.06	**0.14**	**0.91**	15.4	16.9
8	sadiobjects	828	0.66	**0.49**	**2.53**	4.42	1.75
9	Microbiota	868	6.36	**0.19**	**0.97**	17.9	18.5
10	mfoem.emotion	902	35.1	**0.88**	**2.77**	42.1	15.2
11	onsumption	945	233	**1.21**	**2.19**	20.8	9.51
12	emistrycomplex	1,041	12.4	**0.61**	**8.53**	14.2	1.66
13	nskisimple	1,737	36.3	**0.21**	**2.9**	29.3	10.1
14	Earthquake	2,013	20.5	**0.68**	**7.73**	14.8	1.91
15	geolOceanic	2,324	23.8	**0.55**	**1.38**	12.1	8.72
16	landCoastal	2,660	29.0	**0.70**	**1.48**	17.2	11.6
17	mergedobi	2,638	351	**0.96**	TO	**364**	-
18	00350	2,638	441	**3.25**	**28.2**	310	11.0
19	obi_functional	2,750	336	**2.49**	**35.3**	342	9.72
20	quanSpace	2,999	145	**0.42**	**38.2**	380	9.95

TABLE 6.10: Time Metrics of tested OWL ontologies using parallel framework with Hermit (timeout (TO) = 1000 seconds) (Sequ = Sequential, Para = Parallel)

#	Ontology	Concept	Precomputing		Classification		Speedup
			Sequ	**Para**	**Para**	**Hermit**	**Factor**
21	EnergyFlux	3,008	193	**0.36**	**121**	277	2.29
22	stateEnergy	3,018	131	**0.99**	**12.2**	72.9	5.95
23	rDataModel	3,049	136	**1.32**	**68.3**	757	11.1
24	virControl	3,274	164	**0.42**	**45.6**	439	9.65
25	aksmetrics	3,889	6.43	**0.6**	**3.25**	13.6	4.20
26	microbial.type	4,636	304	**0.51**	**26.6**	308	11.6
27	MSC_classes	5,559	156	**1.46**	TO	TO	-
28	obo.PREVIOUS	6,580	378	**0.55**	**311**	646	2.07
29	obo.CURRENT	6,595	391	**0.74**	**112**	452	4.02
30	PREVIOUS	7,335	TO	**1.79**	TO	TO	-
31	SMOtop	7,782	TO	**32.4**	**432**	TO	2.31
32	COSMO	7,804	TO	**33.4**	**728**	TO	1.37
33	compatibility	7,929	37.7	**0.63**	**20.1**	22.2	1.10
34	EnzyO	8,223	TO	**1.74**	TO	TO	-
35	natural.product	9,463	67.9	**2.16**	98.7	**11.2**	0.11
36	vertebrate	18,092	TO	**13.8**	TO	TO	-
37	temetazoan	32,750	TO	TO	TO	TO	-
38	ewasserted	63,848	TO	**5.3**	TO	TO	-
39	ersections	70,232	TO	TO	TO	TO	-
40	havioredit	99,399	TO	TO	TO	TO	-

7 Conclusion

The main purpose of this research is to design and implement a sound and complete parallel framework for ontology classification. This framework can be used for different DL reasoners to speedup their classification process. Following all the objectives described in Chapter 1, a novel prototype of parallel classification approach was developed and a series of experimental results were conducted by using selected real world ontologies, which demonstrated that this work can be adapted to the various ontologies. These ontologies are not only complex but also have an increasing number of concepts in order to better compare our approach with existing sequential DL reasoners.

7.1 Thesis Contributions

- The first version of our parallel approach (see Chapter 4) has been published in [46] and [47] which presents a thread-level parallel architecture for ontology classification and ideally suited for shared-memory SMP servers, but does not rely on locking techniques and thus avoids possible race conditions. A newly designed atomic data structure (see Section 4.4.1) consists of a possible list and remaining list for all the satisfiable concepts of an ontology to record all the subsumption relations. There are two parallel strategies: random division and group division strategies (see Section 4.3), which have been defined and applied in this

approach. The prototype is evaluated with a set of real-world ontologies. The results demonstrate a very good scalability resulting in a speedup that is linear to the number of available cores.

- The improved version of this approach [48] (see Chapter 5) can be applied to existing OWL reasoners and speed up their classification process. The data structure is updated with subsumee, equivalent and disjoint sets in Section 5.2. The parallel precomputing phase (see Section 5.3.1) is applied to speed up the classification process. A new work-stealing strategy (see Section 5.3.2) is designed to reduce the overhead. In comparison to the selected black-box reasoner our results demonstrate that the wall clock time of ontology classification can be improved by one order of magnitude for most real-world ontologies.

- Ontology Scale. For the first evaluation of our parallel classification architecture (see Section 6.1), a set of 9 real-world ontologies are selected that contain up to 13,000 concepts and 33,000 axioms to test scalability. For the small-sized the peak is reached around 140-200 workers and large-sized around 200-280 workers. With a growing ontology size, a better speedup can be achieved by increasing the number of workers, which is linear to the number of threads.

- Ontology Complexity. For the first evaluation (see Section 6.1), a set of 6 real-world ontologies are selected that contain up to 7,000 axioms and 967 qualified cardinality restrictions (QCRs). As the number of threads is increased, a better speedup is observed and the maximum is reached with 60-100 threads except for the one with 446 QCRs, which has small-sized concepts and reached its maximum speedup around 40 threads.

- Runtimes. For the improved version (see Section 6.2), 11 different ontologies with QCRs are selected and compared with black-box reasoners over the whole

classification process. The evaluation shows that our approach is more efficient in most cases for ontologies which have less than 10,000 concepts with up to 120 threads. The evaluation results indicate that if our framework would use a different and more efficient black-box reasoner, it could scale better for more difficult and/or bigger ontologies.

7.2 Future Work

The main parts of this work have been finished. Furthermore, there are some thoughts and proposals which can be discussed and explored by further experiments.

- Shared-memory Structure. In this research, a shared-memory half-matrix data structure is designed to reduce the exchange of updates and requirements of memory. However, through the observation of conducting experiments, it is difficult to find an ideal server which has a shared-memory with enough resources for processing classification of much bigger size ontologies as we expected. Therefore, a distributed memory approach might be a better option for further research to enlarge the scalability of much bigger size ontologies.

- The Black-box Reasoners. This work has been successfully implemented with two DL reasoners: Hermit and JFact. In comparison to the selected black-box reasoners our results demonstrate that the wall clock time of ontology classification can be improved by one order of magnitude for most real-world ontologies. Due to the wide range and different characteristics of DL reasoners, a module which combines different reasoners can be considered to improve the whole classification process [49].

- Optimization Techniques. Currently, the techniques applied in this research rely on parallel computing and transitive closure to reduce more subsumption relations without testing and schedule all the available threads to reduce the overhead. Due to the features of different modern techniques, more strategies from different perspectives could be applied to simply the classification process, such as learning the feasibility of reasoners and the internal relations of an ontology.

Bibliography

[1] Acar, U. A., Blelloch, G. E., and Blumofe, R. D. (2000). The data locality of work stealing. In *In Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, pp. 1-12. ACM.*

[2] Ameloot, T. J., Geck, G., Ketsman, B., Neven, F., and Schwentick, T. (2017). Reasoning on data partitioning for single-round multi-join evaluation in massively parallel systems. In *Communications of the ACM*, volume 60, pages 93–100.

[3] Aslani, M. and Haarslev, V. (2010). Parallel TBox classification in description logics - first experimental results. In *Proc. of the 19th European Conf. on AI*, pages 485–490.

[4] Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D., and eds (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge University Press.

[5] Baader, F., Franconi, E., B. Hollunder, B. N., and Profitlich, H. (1994). An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management 4, 109–132.*

[6] Baader, F. and Sattler, U. (2001). An Overview of Tableau Algorithms for Description Logics. *Studia Logica*, 69:5–40.

[7] Benavides, D., Pablo, T., and Ruiz-Cortés, A. (2005). Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503, Berlin, Heidelberg. Springer.

[8] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

[9] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748.

[10] Borgida, A. and Serafini, L. (2003). Distributed description logics: Assimilating information from peer sources. *In Journal on data semantics I, pp. 153-184. Springer, Berlin, Heidelberg.*

[11] Bruijn, J. D. and Heymans, S. (2007). Logical Foundations of (e)RDF(S): Complexity and Reasoning. pages pp 86–99. 6th International Semantic Web Conference.

[12] Bulpin, J. R. and Pratt, I. (2005). Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*.

[13] Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., and Tsigas, P. (2017). *Programming Multicore and Many-core Computing Systems*, chapter Lock-free concurrent data structures. John Wiley and Sons, Inc.

[14] Chase, D. and Lev, Y. (2005). Dynamic circular work-stealing deque. In *In Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, pp. 21-28. ACM.*

[15] Chen, Y.-K., Tian, X., Ge, S., and Girkar, M. (2004). Towards efficient multi-level threading of h. 264 encoder on intel hyper-threading architectures. In *18th International Parallel and Distributed Processing Symposium, Proceesings, IEEE.*

[16] Davis, E. and Marcus, G. (2016). The scope and limits of simulation in automated reasoning. *Artificial Intelligence (2016)*, 233:60–72.

[17] Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51.1*, 107-113.

[18] Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., and Nieplocha, J. (2009). Scalable work stealing. In *Proc. ACM Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE.

[19] Faddoul, J. and MacCaull, W. (2015). Handling non-determinism with description logics using a fork/join approach. *International Journal of Networking and Computing*, 5(1):61–85.

[20] Gardiner, T., Horrocks, I., and Tsarkov, D. (2006). Automated benchmarking of description logic reasoners. In *Proc. 19th Int. Workshop on Description Logics (DL'06)*, volume 198, page 191.

[21] Glimm, B., Horrocks, I., Motik, B., Shearer, R., and Stoilos, G. (2012). A Novel Approach to Ontology Classification. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14(0):84 – 101.

[22] Glimm, B., Horrocks, I., Motik, B., Stoilos, G., and Wang, Z. (2014). HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269.

[23] González, J. F. (2017). *Java 9 Concurrency Cookbook*. Packt Publishing Ltd.

[24] Guo, Y., Pan, Z., and Heflin., J. (2005). LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182.

[25] Haarslev, V., Hidde, K., Möller, R., and Wessel, M. (2012). The RacerPro knowledge representation and reasoning system. *Semantic Web*, 3(3):267–277.

[26] Horridge, M. and Bechhofer, S. (2011). The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21.

[27] Hsu, C. and Feng, W. (2005). A power-aware run-time system for high-performance computing. In *In SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, pp. 1-1. IEEE.*

[28] Huang, W., Liu, J., Abali, B., and Panda, D. K. (2006). A case for high performance computing with virtual machines. In *In Proceedings of the 20th annual international conference on Supercomputing, pp. 125-134. ACM.*

[29] Kazakov, Y., Krötzsch, M., and Simančík, F. (2011). Concurrent classification of $\mathcal{EL}$ ontologies. In *International Semantic Web Conf.*, pages 305–320.

[30] Kazakov, Y., Krötzsch, M., and Simancik, F. (2012). ELK Reasoner: Architecture and Evaluation. *OWL Reasoner Evaluation Workshop*.

[31] Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann.

[32] Koufaty, D. and Marr, D. T. (2003). Hyperthreading technology in the netburst microarchitecture. In *IEEE Micro 23(2)*.

[33] Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., and Rooholamini, R. (2002). An empirical study of hyper-threading in high performance computing clusters. In *Linux HPC Revolution 45*.

[34] Magro, W., Petersen, P., and Shah, S. (2002). Hyper-threading technology: Impact on compute-intensive workloads. *Intel Technology Journal 6(1)*.

[35] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. (2002). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal, 6(1)*.

[36] Meissner, A. (2011). Experimental analysis of some computation rules in a simple parallel reasoning system for the ALC description logic. *International Journal of Applied Mathematics and Computer Science*, 21(1):83–95.

[37] Mergen, M. F., Uhlig, V., Krieger, O., and Xenidis, J. (2006). Virtualization for high-performance computing. In *ACM SIGOPS Operating Systems Review, 40(2), 8-11*.

[38] Metke-Jimenez, Alejandro, and Lawley, M. (2013). Snorocket 2.0: Concrete Domains and Concurrent Classification. *The 2nd International Workshop on OWL Reasoner Evaluation*.

[39] Möller, R. and Haarslev, V. (2009). Tableau-Based Reasoning. *Handbook on Ontologies-Tableau-based reasoning*, pages pp 509–528.

[40] Motik, B. (2009). Resolution-Based Reasoning for Ontologies. *Handbook on Ontologies-International Handbooks on Information Systems*, pages pp 529–550.

[41] Mutharaju, R., Maier, F., and Hitzler, P. (2010). A MapReduce algorithm for $\mathcal{EL}_{+}$. *23rd International Workshop on Description Logics DL2010, vol. 456*.

[42] Mutharaju, R., Mateti, P., and Hitzler, P. (2015). Towards a rule based distributed OWL reasoning framework. In *Intern. Experiences and Directions Workshop on OWL*, pages 87–92. Springer.

[43] Nieuwpoort, V., V., R., Maassen, J., Wrzesińska, G., Hofman, R. F., Jacobs, C. J., Kielmann, T., and Bal, H. E. (2005). Ibis: a flexible and efficient java-based grid

programming environment. In *Concurrency and Computation: Practice and Experience 17, no. 7-8: 1079-1107*.

[44] ORE (2014). 3rd OWL reasoner evaluation (ORE) workshop.

[45] ORE (2015). 4th OWL reasoner evaluation (ORE) workshop.

[46] Quan, Z. and Haarslev, V. (2017). A parallel shared-memory architecture for OWL ontology classification. In *46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 200–209. IEEE.

[47] Quan, Z. and Haarslev, V. (2018). A parallel computing architecture for high-performance OWL reasoning. *Parallel Computing (2018)*, https://doi.org/10.1016/j.parco.2018.05.001.

[48] Quan, Z. and Haarslev, V. (2019). A framework for parallelizing OWL classification in description logic reasoners. *submitted to International Joint Conferences on Artificial Intelligence, 2019*.

[49] Romero, A. A., Grau, B. C., and Horrocks, I. (2012). More: Modular combination of owl reasoners for ontology classification. In *International Semantic Web Conference, pp. 1-16. Springer, Berlin, Heidelberg*.

[50] Schmidt-Schauß, M. and Smolka, G. (1991). Attributive concept descriptions with complements. *Artificial intelligence 48.1: 1-26*.

[51] Seidenberg, J. and Rector, A. (2006). Web ontology segmentation: analysis, classification and use. In *15th international conference on World Wide Web. ACM*.

[52] Serafini, L. and Tamilin, A. (2005). Drago: Distributed reasoning architecture for the semantic web. In *European Semantic Web Conf.*, pages 361–376. Springer.

[53] Steigmiller, A., Liebig, T., and Glimm, B. (2014). Konclude: system description. *Web Semantics*, 27:78–85.

[54] Suksompong, W., Leiserson, C. E., and Schardl, T. B. (2016). On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100–106.

[55] Tsarkov, D. and Horrocks, I. (2006). FaCT++ description logic reasoner: System description. In *International Joint Conference on Automated Reasoning*, pages 292–297.

[56] Tsarkov, D., Riazanov, A., Bechhofer, S., and Horrocks, I. (2004). Using vampire to reason with owl. In *In International Semantic Web Conference, pp. 471-485. Springer, Berlin, Heidelberg*.

[57] Urbani, J. (2010). Scalable and parallel reasoning in the semantic web. In *The Semantic Web: Research and Applications*, pages 488–492. Springer.

[58] Urbani, J., Kotoulas, S., Maassen, J., Harmelen, F. V., and Bal, H. (18 October 2012). WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web*.

[59] Urbani, J., Kotoulas, S., Oren, E., and Harmelen, F. V. (2009). Scalable distributed reasoning using mapreduce. In *International Semantic Web Conference, pp. 634-649. Springer, Berlin, Heidelberg*.

[60] Wu, K. and Haarslev, V. (2012). A parallel reasoner for the description logic $\mathcal{ALC}$. In *Proc. of the Int. Workshop on Description Logics*, pages 378–388.

[61] Wu, K. and Haarslev, V. (2013). Exploring parallelization of conjunctive branches in tableau-based description logic reasoning. In *Proc. of the Int. Workshop on Description Logics*, pages 1011–1023.

[62] Wu, K. and Haarslev, V. (2014). Parallel OWL reasoning: Merge classification. In *Proc. of the 3rd Joint Int. Semantic Technology Conf. (JIST), Seoul, Korea, November 28-30, 2013*, LNCS, pages 211–227.

Appendix

Publications

1. Zixi Quan and Volker Haarslev. A Parallel Shared-Memory Architecture for OWL Ontology Classification. *46th International Conference on Parallel Processing Workshops (ICPPW)*, Pages 200-209, 2017

2. Zixi Quan and Volker Haarslev. A parallel computing architecture for high-performance OWL reasoning. *Parallel Computing Journal (2018)*. https://doi.org/10.1016/j.parco.2018.05.001 (In Press).

3. Zixi Quan and Volker Haarslev. A Framework for Parallelizing OWL Classification in Description Logic Reasoners. *International Joint Conference on Artificial Intelligence (IJCAI) 2019* (Under Review).