TOWARD WEBSCALE, RULE-BASED INFERENCE ON THE SEMANTIC WEB VIA DATA PARALLELISM

By

Jesse Weaver

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

James A. Hendler, Thesis Adviser

Christopher Carothers, Member

Peter Fox, Member

David Mizell, Member

Rensselaer Polytechnic Institute
Troy, New York

February 2013
(For Graduation May 2013)

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
FEB 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Toward Webscale, Rule-Based Inference on the Semantic Web Via Data Parallelism	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Rensselaer Polytechnic Institute,110 Eighth Street,Troy,NY,12180**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis considers the problem of scaling rule-based inference to large quantities of RDF data found on the Semantic Web. The general approach is one of data parallelism, that is, dividing data among processors such that the collective results of each processor's individual inference is the same as though inference was performed sequentially. In this way, theoretically speaking, more processors can be added to accommodate more data. The problem is rst considered from the perspective of the operational semantics of inference with production rules. The question is asked, under what conditions is embarrassingly parallel inference guaranteed to be correct? Su cient conditions are determined and proven at both a ne-grained level close to the basic operational semantics and a more coarse-grained level that applies directly to rules. The con- ditions are placed on the relationship between rules and distribution schemes, that is, the way in which data is assigned to processors. Then, a special class of distribution schemes is considered called replication schemes. Replication schemes require that individual data either be replicated to all processors or placed arbitrarily on some processor(s). The aforementioned con- ditions are then reformulated to consider replication schemes which reveals that testing the conditions for replication schemes is reducible to satis ability (SAT) and not only SAT but 2SAT. An augmented version of this reduction which is a reduction to 3SAT also accounts for the possibility to eliminate some rules in order to improve parallelization. These reductions along with a proposed methodology for restricting rules are used to derive restricted versions of the RDFS and OWL2RL rules that are amenable to parallel inference. Finally, an evaluation is performed that tests these theoretical ndings for re- stricted versions of RDFS and OWL2RL inference on two large, well-known datasets exceeding a billion triples: LUBM10K and BTC2012. The LUBM10K dataset repre- sents an optimistic case, meaning that if performance is poor with LUBM10K, then it will likely be poor on many datasets. On the other hand, the BTC2012 dataset represents a pessimistic case, meaning that if performance is good with BTC2012 then it is likely that performance will be good with other datasets. While the usual scalability metrics are used (speedup, e ciency, etc.), the Karp-Flatt metric reveals that inference is almost entirely parallel for LUBM10K data, demonstrating the practical feasibility of the theoretical ndings. However, for BTC2012, it must be ensured that there is su cient memory and load-balancing to achieve this high level of scalability on distributed memory architectures. Regardless, for feasible cases very low times are achieved for LUBM10K (seconds) and BTC2012 (minutes).

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**201**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

Acknowledgement must be given first and foremost to President Shirley Ann Jackson and Professor Morris A. Washington for so generously funding the first Patroon Fellowship which funded the vast majority (four years) of my graduate education. A most heartfelt thank you.

This research was sponsored in part by a grant from DARPA's Transformational Convergence Technology Office and also by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Army Research Laboratory or the U.S. Government.

Thank you to James A. Hendler, Christopher Carothers, Peter Fox, and David Mizell for participating in my doctoral dissertation committee.

Thank you to the helpful staff at the Computational Center for Nanotechnology Innovations – Christopher Carothers, Angie Nunez, Jackie Stampalia, Lindsay Todd, Adam Todorski, Daniel M. Weeks – for their support in using their facilities over the years.

Thank you to my fellow graduate students with whom I have collaborated and had interesting discussions, particularly Gregory Todd Williams, Medha Atre, Ankesh Khandelwal, and Xian Li.

Thank you to James A. Miller and David Petrovich for introducing me to the Semantic Web. The work we attempted together was the initial inspiration for this research topic.

Thank you to my parents Rebecca and Carmo Weaver. Thank you to Xianhua Zhu and Jingxiang He who, in many ways, have treated me as their own son. Special thanks to Rebecca Weaver and Xianhua Zhu for their active care of my children, Rosemary and David Weaver.

Thank you to Melba and John Elvin who showed my family extraordinary kindness in a time of hardship, treating us like their own family.

Thank you to Tanya and James Glasscock who were there when nobody else was.

Thank you to Joshua Weaver for visiting.

Thank you to Matthew Weaver for listening and being alert to my situation.

Thank you to William Tullis Lambert for his light at the darkest time.

Thank you to Guangyu Christina Zhu-Weaver, a wonderful wife, a marvelous mother, my partner in life, and my best friend.

ABSTRACT

This thesis considers the problem of scaling rule-based inference to large quantities of RDF data found on the Semantic Web. The general approach is one of data parallelism, that is, dividing data among processors such that the collective results of each processor's individual inference is the same as though inference was performed sequentially. In this way, theoretically speaking, more processors can be added to accommodate more data.

The problem is first considered from the perspective of the operational semantics of inference with production rules. The question is asked, under what conditions is embarrassingly parallel inference guaranteed to be correct? Sufficient conditions are determined and proven at both a fine-grained level close to the basic operational semantics and a more coarse-grained level that applies directly to rules. The conditions are placed on the relationship between rules and distribution schemes, that is, the way in which data is assigned to processors.

Then, a special class of distribution schemes is considered called replication schemes. Replication schemes require that individual data either be replicated to all processors or placed arbitrarily on some processor(s). The aforementioned conditions are then reformulated to consider replication schemes which reveals that testing the conditions for replication schemes is reducible to satisfiability (SAT), and not only SAT but 2SAT. An augmented version of this reduction which is a reduction to 3SAT also accounts for the possibility to eliminate some rules in order to improve parallelization. These reductions along with a proposed methodology for restricting rules are used to derive restricted versions of the RDFS and OWL2RL rules that are amenable to parallel inference.

Finally, an evaluation is performed that tests these theoretical findings for restricted versions of RDFS and OWL2RL inference on two large, well-known datasets exceeding a billion triples: LUBM10K and BTC2012. The LUBM10K dataset represents an optimistic case, meaning that if performance is poor with LUBM10K, then it will likely be poor on many datasets. On the other hand, the BTC2012 dataset

represents a pessimistic case, meaning that if performance is good with BTC2012, then it is likely that performance will be good with other datasets. While the usual scalability metrics are used (speedup, efficiency, etc.), the Karp-Flatt metric reveals that inference is almost entirely parallel for LUBM10K data, demonstrating the practical feasibility of the theoretical findings. However, for BTC2012, it must be ensured that there is sufficient memory and load-balancing to achieve this high level of scalability on distributed memory architectures. Regardless, for feasible cases, very low times are achieved for LUBM10K (seconds) and BTC2012 (minutes).

CHAPTER 1
INTRODUCTION

Since its conception, the World Wide Web (Web) has presented a persistent problem: given the vast amount of information published, how does one find anything of a particular interest? Traditionally, the Web has been viewed as a set of interlinked documents in which the documents are – from the perspective of algorithms – "bags of words." Such a view of the Web renders the information meaningless to machines, compelling the need for algorithms that ignore the inherent meaning of the information and instead rely on probabilities or cleverly devised ranking metrics based on structural characteristics of the Web. Modern search engines have essentially solved this particular problem.

The Semantic Web [1] – the Web as it is today – contains not only "bags of words" but also structured data with explicit semantics, in the form of Resource Description Framework (RDF) [2] triples. These explicit semantics allow algorithms to discover, in some sense, the precise meaning of the data. Utilizing this information to improve search and data integration is still an open problem. While the structured data has an explicit semantics, these semantics often imply other data. In other words, additional data need to be inferred to more fully reveal the explicit data's meaning and entailments.

It is arguably the case that the most common way of semantically enriching data on the Semantic Web is by providing an ontology in the Web Ontology Language (OWL) [3] that describes the data. OWL is based on Description Logic (DL) [4] which has the desirable property of decidability. However, this characteristic significantly limits expressivity, and so rule-based inference has been turned to as an alternative. This is particularly evidenced by the recent World Wide Web Consortium (W3C) Recommendation of the Rule Interchange Format (RIF) [5].

The topic of rule-based inference has a long research history rooted in artificial intelligence (particularly logic programming and theorem proving) and databases. Rules can be loosely thought of as If-Then statements in which some condition

implies some consequence. Rules can be roughly categorized into logic rules and production rules (or more generally, rules with actions). This proposal focuses on production rules, which are capable of expressing some other forms of rules like Datalog [6].

The general approach to scaling inference to larger amounts of data has been to employ parallelism. Scalability of a system is determined based on the kind of scaling under consideration, traditionally strong scaling in which execution time should decrease as number of processors increases, holding workload fixed. In strong scaling, the ideal[1] case is that $T_N = \frac{T_1}{N}$ where T_1 is the execution time on a single processor, N is a positive number of processors, and T_N is the execution time on N processors. A more appropriate notion of scalability for the Semantic Web is data scaling [7] in which the ratio of dataset size to number of processors is fixed, and execution time is desired to remain constant. In this case $\mathcal{T}_N = \mathcal{T}_1$ is the ideal case, where $\mathcal{T}_1$ is the execution time on a single processor at capacity, and $\mathcal{T}_N$ is the execution time on N processors at capacity. The reason for this alternative perspective on scalability is that handling the growth of data is the primary challenge, and reducing execution time is secondary. Either way, let τ_N denote the execution time on N processors, and let τ_* be its ideal value. In this way, the following arguments can be made independent of the notion of scalability. $\tau_N > \tau_*$ is an indication that there is some performance cost to parallelizing the computation. That is, in practice, $\tau_N = \tau_* + cost(N)$.

Traditional research in parallel, rule-based inference (mostly from the late 1980s to mid-1990s) focused on the general case, making no assumptions about the rules or data [8, 9, 10, 11, 12, 13, 14, 15, 16]. Handling large amounts of data was considered a problem of providing correct inference while minimizing $cost(N)$ by improving load-balancing and reducing communication. Some approaches required elimination of redundant work [12, 13].

However, compared to the focus of traditional research, the Semantic Web is a relatively special case in which the amount of data far exceeds the number of rules,

[1]Ideal in a theoretical sense. In practice, it is possible for T_N to be smaller if superlinear speedup is achieved by spreading out the data enough so that each processor can better utilize cache.

and even without consideration of data, the number of rules can be relatively few as in the RDF Schema (RDFS) [17] rules. This change in focus is expressed in the cleverly coined hypothesis of the Semantic Web originating with co-inventor James Hendler: "a little semantics goes a long way" [18]. In the context of this work, the "little semantics" is the few rules, and the "long way" is the amount of data.

As a result, some recent works in parallel, rule-based inference on the Semantic Web have taken a different perspective of parallelism than in traditional research on parallel, rule-based inference. In the presence of few rules and relatively small ontologies, good scalability can sometimes be achieved simply by replicating portions of the data to all processors and allowing them to perform inference independently, that is, in an embarrassingly parallel fashion [19, 20, 21]. Therefore, except for possible contention during the initial data distribution, $cost(N)$ reflects only data-dependent costs like load imbalance caused by skew in data distribution, or redundant work caused by replication of (initial or inferred) data. Therefore, assuming each processor runs the best sequential inference system, $cost(N)$ is purely determined by data distribution. Thus, determination of a distribution scheme is important and leads to the first contribution.

> **The first, novel contribution of this thesis is to determine sufficient conditions under which a data distribution scheme supports correct (embarrassingly) parallel inference for a given set of rules, independent of features of the data.**

Previous works on rule-based inference on the Semantic Web have typically placed restrictions on the data [19, 20, 21] in order to achieve complete parallel inference with some fixed set of rules. This work instead makes no assumptions about the data.

In many interesting cases, though, the only solution is to replicate all the data to all processors, which defeats the purpose of parallelization (which is to either grow the dataset beyond the capacity of a single machine and/or improve performance). In such cases, one might consider sacrificing some semantics of the program (rules) to achieve better parallelization. This leads to the next contribution.

The second, novel contribution of this thesis is a method – partially formal and partially heuristic – that can be used to restrict a set of rules such that the restricted version is amenable to parallelization.

This second contribution is achieved by considering a special case of data distribution which allows the conditions for parallelization to be reduced to the satisfiability problem (SAT). Combined with an intuition-guided methodology and a few optimization heuristics, this approach is used to derive restricted versions of the RDFS and OWL 2 Rule Language (OWL2RL) [3] rules that are amenable to parallelization.

The third, novel contribution of this thesis is a performance evaluation of parallel inference using restricted RDFS and OWL2RL rule sets to empirically test the practical value of the aforementioned theoretical findings.

The evaluation is run on a large symmetric multiprocessing (SMP) machine and a Blue Gene/Q [22]. The datasets used in the evaluation are the synthetic Lehigh University Benchmark (LUBM) [23] dataset (unrealistically optimistic, best case scenario) and the 2012 Billion Triples Challenge dataset (BTC2012, an abysmally difficult, worst-case scenario) [24]. Performance metrics include execution time, relative speedup, efficiency, the Karp-Flatt metric [25], and the recently proposed growth efficiency [7]. It is found that a high degree of scalability is achievable with these restricted rule sets when there is sufficient memory and load-balancing.

CHAPTER 2
HISTORICAL REVIEW

There are three main categories for historical review: distributed logic programs, parallel production rule systems, and recent work in scalable, rule-based inference on the Semantic Web. Before discussing related works in these particular fields, though, the taxonomy of parallel strategies of deduction given by Bonacina [26] is discussed to give context to the strategies employed in previous work. Kotoulas et al. [27] similarly cover the discussion from section 2.1 and the review of related works in section 2.4.

2.1 Taxonomy of Parallel Deduction

Bonacina [26] describes three levels of parallelism differing in granularity: term-level parallelism (fine-grained), clause-level parallelism (mid-grained), and search-level parallelism (course-grained). Term-level parallelism seeks to parallelize frequent, low-level operations such as term rewriting and unification. For example, unifying terms $p(a_1, \ldots, a_m)$ and $q(b_1, \ldots, b_n)$ consists of checking to make sure $p = q$, $m = n$, and then attempting to unify each a_i and b_i for $1 \leq i \leq n$. If there exists j, k such that $j \neq k$ and there is no dependency between a_j and a_k, and there is no dependency between b_j and b_k, then a_j and b_j can be unified independently of a_k and b_k, which means the two unifications can be executed in parallel.

Clause-level parallelism seeks to parallelize individual inference steps. This could be parallelizing search for satisfactions of different subgoals in a rule, or it could be parallelizing search for satisfactions of the same subgoal. This is discussed more in section 2.2.1.

Search-level parallelism seeks to divide the entire search space among processors. Search-level parallelism has the distinguishing characteristic that its coarse-grained nature lends itself to distributed environments. However, such parallelism is hardly as straightforward as the previous two. The manner in which the search space is divided depends on the goal to be achieved and the means by which it is to

be achieved. There are two axes to search-level parallelism: whether the process is homogeneous or heterogeneous, and whether multi-search or distributed search (or both) are used.

In homogeneous, search-level parallelism, all processors use the same inference system. For example, they all use a DL reasoner, or they all use a Prolog engine. In contrast, heterogeneous systems combine different inference systems.

In multi-search, search-level parallelism, processors may employ different search plans. In other words, even if each processor has the same inference system (e.g., a DL reasoner), each processor may apply rewrite rules in different orders or to different formulas. The main point is that the processors may handle non-determinism differently. On the other hand, distributed search has nothing to do with non-determinism (although it does not preclude multi-search). The defining characteristic of distributed search is that processors differ in distribution of the problem, for example, by data and/or rule distribution. Distributed search often requires the processors to communicate in order to ensure completeness or improve load-balancing.

Thus, search-level parallelism can differ by diversity of inference systems, search plans, and/or problem distribution. Since it allows for distribution of data, search-level parallelism is the form of parallelism focused on herein, specifically homogenous, non-multi-search, distributed search. In other words, I am seeking to achieve parallelism solely by means of data distribution. Each processor is assumed to have the same inference system and the same search plan (i.e., same way of handling non-determinism).

2.2 Distributed Logic Programs

Parallelism for improving the performance of reasoning with logic programs has been well-studied as shown in the survey papers [10, 26]. This section considers the most relevant logic programming, namely Prolog [10] and Datalog [6].

2.2.1 Prolog

Strategies for parallel, backward-chaining inference in Prolog have been surveyed by Gupta et al. [10]. Prolog rules have the form H `:-` $B_1, \ldots, B_n$, where H

and each B_i are atomic formulas. H is called the head or consequent, and $B_1, \ldots, B_n$ is called the body or antecedent. All variables are implicitly universally quantified to the scope of the rule. Gupta et al. delineate three forms of parallelism: unification parallelism, and-parallelism, and or-parallelism.

Unification parallelism is a form of term-level parallelism, an example of which has already been given in section 2.1. And-parallelism is concerned with satisfying the subgoals (B_i) in the body of a rule in parallel. Doing so is not necessarily straightforward since subgoals in the body often share variables, and thus, they cannot be satisfied independently. In contrast, or-parallelism certainly allows for independent processing. If a (sub)goal can be unified to the heads of multiple rules, then search for satisfactions of these rules can be performed independently for each rule. If the goal is the query being posed to the Prolog engine, then or-parallelism in this case is search-level parallelism. If the goal is a subgoal of a rule, then or-parallelism in this case is clause-level parallelism.

Note that of these approaches, none of them use a distributed search strategy. Thus, these forms of parallelism are orthogonal to the focus of this work.

2.2.2 Datalog

Unlike Prolog, Datalog has a significant history of distributed search. Relevant works and strategies have been summarized in [11]. The main approaches have been *program restriction* and *predicate decomposability*.

The goal of program restriction is to effectively distribute the firing of rule instances (i.e., the drawing of individual inferences from specific rules) to processors by appending some conditions to the rule bodies. Such an approach usually uses some hash-distribution scheme for assigning facts to processors, effectively implementing a distributed hash-join. Inferences then need to be communicated between processors to ensure completeness; in other words, inferences are also hash-distributed. Attempts to optimize restricted programs is by way of minimizing communication. Restricted programs are an example of distributed search. Unlike embarrassingly parallel inference, program restriction simply assumes communication between processors (although it does not preclude the possibility of restricting a program such

that no communication is necessary).

In contrast, predicate decomposability as introduced by Wolfson and Silberschatz [13] aims to distribute data to processors such that embarrassingly parallel inference for a given predicate is correct and each processor's closure for that predicate is disjoint from every other processor's closure. (Closure is the set of all possible inferences for the given facts and rules, which is always derivable in Datalog). Thus, it is also an example of distributed search. In [13], Wolfson and Silberschatz characterized three types of single rule programs (sirups) for which predicate decomposability is achievable. Later, Wolfson and Ozeri [12] added an additional criterion, that each processor must have at least one fact with the predicate being decomposed. In the same paper, they present a theorem with sketch proof stating that it is undecidable whether a program is decomposable on a given predicate.

In contrast to predicate decomposability, the work proposed herein seeks only to achieve soundness (implied in Datalog) and completeness; there is no strict requirement of disjointness or whether each processor has at least one fact for some predicate. The logical desirability behind the disjointness criterion is that if each inference is drawn by exactly one processor, then no redundant work is performed, and assuming perfect load-balancing, a high parallel efficiency can be achieved. This is a very strict requirement, one that surely cannot be met in many cases.

2.3 Parallel Production Rule Systems

Unlike with (typical recursive) Datalog, Production Rule Systems (PRSs) allow functions, negation of formulas, and retraction of facts.[2] Therefore, inference in a PRS may not be decidable, and the order in which rule instances (rules with bound variables) are fired may have an impact on which facts are inferred and when. A mechanism called a conflict resolution strategy (CRS) is used to define how rule instances are fired.

PRSs follow a cycle of match, resolve, and fire. First, rules are matched to the knowledge base (factbase, set of facts) to derive all possible rule instances, then the

[2]Variants of Datalog also allow forms of negation, but to the best of my knowledge, they have been generally unstudied in the context of parallelism.

CRS determines which rule instances should be fired and in what order, and then the selected rules are fired. Most work in parallel PRSs center around parallelizing the OPS5 PRS [28] which used the well-known RETE algorithm [29] for the match phase [8]. Much of the work in parallelizing OPS5 has focused on the match phase since it accounted for the majority of execution time [9]. Other work has focused on parallelizing rule firing, a much more difficult problem. If resolution and rule firing can be parallelized, then the entire inference cycle can be parallelized.

Works on parallelizing the match phase have focused on parallelizing the RETE algorithm on different architectures. The main advantage of RETE was that it could quickly match many rules ("productions"), but this advantage is of less importance in the present landscape given the change of focus from many rules to much data [8]. Parallelizing the match phase with few rules is well understood, particularly given the vast research on parallelizing relational queries (to which rule conditions can typically be reduced).

This thesis focuses on parallelizing the entire inference cycle; therefore, it is more important to consider previous work toward that end. The difficulty in parallelizing the inference cycle is in simultaneous firing of rules such that the effect is somehow similar to firing the rules under a sequential CRS. In OPS5, in each inference cycle, only a single rule instance can be selected for firing [28], and therefore, there is a deterministic order in which individual rule instances are fired. Given this context, Schmolze proposed the *serializability criterion* for correctness of parallel rule firing:

> "We say that the coexecution of many instantiations is serializable if and only if there exists some serial execution that would produce the same result using the same instantiations." [14]

In other words, under the serializability criterion, the result of firing multiple rule instances in parallel must be the same as firing the same instances in *some* valid order under the sequential CRS. Schmolze argues that ensuring serializability is important because developers rely on the CRS to understand and enforce the semantics of their programs (rules).

To satisfy the serializability criterion, Kuo and Moldovan [9] define two problems that, when solved, ensure serializability.

- The *compatibility problem* is the problem of firing incompatible rule instances. Conditions are given by Kuo and Moldovan for when rule instances are considered compatible. One is that the firing of one rule instance before another should not preclude the firing of the latter rule instance (e.g., one rule instance retracts a fact that is used to match the condition of the other rule instance). Another is that the actions of two rule instances should not be contradictory (e.g., one rule instance asserts a fact that the other retracts).

- The *convergence problem* is the problem of ordering the individual, parallel rule firings such that they are serializable. Ensuring compatibility simply means that a single, simultaneous firing of (compatible) rule instances will have a valid serialization. However, it does not guarantee that the ordering of such simultaneous firings will have an *overall* valid serialization. In other words, arbitrarily chaining together two valid serializations does not necessarily create a single valid serialization. This is the convergence problem.

The compatibility problem makes particular sense for the OPS5 PRS which requires that only a single rule instance be selected for firing in any given cycle. Thus, the firing of one rule instance can indeed cause another (unfired) rule instance to be unmatched in subsequent inference cycles. That is, it is actually possible for the firing of a rule instance to preclude the firing of another rule instance. If the requirement that only a single rule instance be fired per cycle is relaxed, then the compatibility problem becomes less of an issue. This will be further addressed in the next chapter.

Concerning the convergence problem, one possible solution observed by Kuo and Moldovan [9] is to simply replace the semantics of a sequential CRS with that of a parallel CRS. The benefits of such an approach is that a parallel CRS can still provide some semantics on which a developer may rely, argued to be essential by Schmolze [14]. However, the burden of defining the semantics of the parallel CRS has commonly been relegated to developers to define so-called "metarules" that

control the parallel execution of the PRS [8].

Another approach to solving the convergence problem (as observed by Kuo and Moldovan [9]) is to introduce non-determinism. If the order of firing rule instances (or some set of rule instances) is completely non-deterministic, then any ordering satisfies the serializability criterion.

This characteristic – that any ordering produces correct results – is actually a strong criterion called the *commutativity criterion* proposed by Ishida and Stolfa [15]. In the case of a completely non-deterministic CRS, ordering of rule instances can be arbitrary, and thus any parallel execution can be considered correct. However, outside of such cases, the developer must prove to him/herself that any ordering of rule instances would produce "correct" results. As pointed out by Amaral and Ghosh [16], the commutativity criterion is generally considered too strict.

Returning to the serializability criterion and non-deterministic CRSs, there have been multiple approaches to introducing non-determinism, but they can be roughly classified into two cases: providing mechanisms to the developer to effectively direct non-determinism (i.e., introduce some determinism), and to divide rules into sequential and parallel sets. Sequential sets are always executed sequentially, but parallel sets are executed either non-deterministically (arbitrary ordering) or simultaneously (similar to inflationary semantics in Datalog with negation [6]). These rulesets are determined by developers, and in some cases, there is a control flow between active rulesets (called "contexts") that provides some determinism.

These approaches may have seemed practical when developers were the only users under consideration. However, given the broad audience of the (Semantic) Web, it seems unduly onerous to require relatively lay people to understand so many fine details. The most practical approach seems to be the formulation of a parallel CRS. A class of CRSs amenable to parallelization is defined in chapter 3.

2.4 Scalable, Rule-based Inference on the Semantic Web

Related works in this area are relatively recent. Perhaps one of the earliest works, from 2006, is toward partitioning OWL knowledge bases for distributed reasoning by Guo and Heflin [30], although it is focused on DLs rather than rules.

Liebig and Muller [31] as well as Bock [32] considered similar problems.

Regarding parallel *rule inference*, though, the earliest works specific to the Semantic Web seem to go back to 2008, focusing exclusively on distributed memory architectures with the exceptions of [33, 34]. Soma and Prasanna considered rule partitioning and data partitioning approaches to parallel OWL Horst inference [35]. OWL Horst is an established, *de facto* standard set of rules based on the pD^* fragment [36, 37] of OWL 1 [38]. Kaoudi et al. presented work on RDFS inference on distributed hash tables (DHTs) [39]. At the 2008 Billion Triples Challenge, Anadiotis et al. presented MaRVIN, a system for sound and eventually complete RDFS inference on clusters that won third place [40], which also lead to subsequent research in 2009 by Oren et al. [41, 42]. MaRVIN was different from previous (and even subsequent) approaches in that it did not necessarily guarantee completeness, although evaluation suggested that it would at least come very close, particularly when using suitable heuristics. Another work distinct from its peers is the work on approximate reasoning by Rudolph et al. [34] in which multiple inference systems were combined not to decrease latency but to increase availability and quality of answers for any given amount of execution time. This is an example of non-distributed, multi-search, as opposed to all other works presented in this section, which are distributed, non-multi-search.

In 2009, Hendler and I [21] demonstrated that certain characteristics of the RDFS rules allow RDFS inference to be performed in an embarrassingly parallel fashion and still achieve soundness and completeness *if* the data met some common conditions (e.g., the `rdfs:subPropertyOf` property has no subproperties), the results of which were used in the system that was part of the champion submission to the 2009 Billion Triples Challenge by Williams et al. [43, 44]. At the same conference, Urbani et al. presented similar findings regarding parallel RDFS inference in a MapReduce framework [20]. Quite recently, Patel-Schneider [45] expanded on the details of the assumed conditions by Hendler and me, and by Urbani et al., showing that producing the *truly* complete finite RDFS closure (no assumed conditions on the data) is inherently serial, where here, "serial" is meant in the theoretical sense, that is, not being *completely* parallel.

In 2010, Urbani et al. extended their work to OWL Horst inference [46], achieving the largest closures on both real-world (on the order of billions of triples) and synthetically generated datasets (100 billion triples) to date. Kotoulas et al. utilized previous MaRVIN work to address the issue of load-balancing in parallel RDFS inference caused by data skew [47]. Hogan et al. presented optimizations for distributed, rule-based reasoning using a subset of the OWL2RL rules [19]. This work by Hogan et al. is particularly interesting for two reasons. First, the inference system disallows inferences resulting from "ontology hijacking" [48], an apparent misuse of ontologies. Second, this work mathematically derives conditions for soundness and completeness for rule-based reasoning (specifically Datalog rules with focus on linear recursion) with ontologies. Goodman and Mizell applied the findings of [20, 21] to implement and optimize RDFS inference on a Cray XMT [33], the first work to make significant use of a shared memory architecture. This system also contributed to the runner up submission to the 2010 Billion Triples Challenge [49].

All of these previous works on rule-based inference on the Semantic Web had been forward chaining reasoners, with the exception of [39]. In 2011, Urbani et al. presented a parallel system for backward-chained inference with RDFS and OWL Horst rules [50]. The collective works of Urbani to date can be found in his recently defended doctoral dissertation [51].

In 2012, Heino and Pan demonstrated parallel RDFS inference on graphics processing units (GPUs) [52].

CHAPTER 3
SUFFICIENT CONDITIONS FOR PARALLEL INFERENCE

In 2009, Hendler and I showed that a restricted version of the RDFS closure (the full closure of which is known to be undecidable [37, 45, 53]) can be produced in an embarrassingly parallel fashion, allowing a high degree of scalability to be achieved by increasing the number of processors with the amount of data [21]. The formal basis for this finding was based on inspection of the RDFS rules and the common assumption that the terminological (then referred to as "ontological") triples constitute a small part of the overall data. In this chapter, I further develop these notions beyond specific application to RDFS toward general application to production rules.

First, it must be defined exactly what is meant by rules. In section 3.1, the notion of rules is defined as a subset of the Production Rule Dialect of RIF (RIF-PRD) [54] with equivalent operational semantics. Then in section 3.2, formal definitions are given for parallel inference, including various definitions for what it means for parallel inference to be "correct". In section 3.3, sufficient conditions are proven for correctness of parallel inference, which is the main contribution of this chapter.

3.1 Rules and their Semantics

In this section, the syntax for rules and the operational semantics of inference are defined. The basis for these definitions are RIF-PRD [54], and as such, much of the content of this section is directly derived from that work. However, the definitions are not exactly the same. Not only have they been reworded and reorganized to fit the language and purpose of this work, but also the definitions herein are more restrictive. Some of the restrictions are merely syntactic (e.g., forcing normal form) while others actually reduce the expressive power of the rule language. Footnotes are provided with information regarding the differences, although they are not necessarily an exhaustive accounting of the differences.

Additionally, mathematical notation will periodically be defined in distinct text blocks to make clear the express intent to use a particular notation throughout the remainder of this work.

3.1.1 Syntax and Notation

Definition 1. *Reserved symbols* are any of the following: `=`, `#`, `##`, `->`, `[`, `]`, `(`, `)`, `List`, `If`, `Then`, `Do`, `Assert`, `Retract`, `And`, `Not`, `External`.

Definition 2. A *variable* is a symbol that is not a reserved symbol and is represented syntactically as a string in typewriter text that does not contain whitespace and that begins with a question mark. For example, `?v`.

Definition 3. A *constant* is a symbol that is not reserved and not a variable represented syntactically as "*lex*"^^<*datatype*> where *lex* is an appropriately escaped string called the lexical representation and *datatype* is an Internationalized Resource Identifier (IRI) [55] identifying how *lex* should be interpreted. Constants are divided into four disjoint sets: (plain) predicate names, built-in (predicate) names, function names, and individual names. Each non-individual name has an associated non-negative integer referred to as its arity. For a non-individual name p, let $arity(p)$ denote the arity of p.

Notation. *Constants with certain datatypes are allowed a syntactic shorthand, as illustrated by example in the following.*

- *"http://www.rpi.edu/"^^<http://www.w3.org/2007/rif#iri> can equivalently be represented <http://www.rpi.edu/> or in appropriate cases, using a Compact URI (CURIE) [56];*
- *"label"^^<http://www.w3.org/2007/rif#local> can equivalently be represented _label;*
- *"1"^^<http://www.w3.org/2001/XMLSchema#integer> can equivalently be represented 1.*

Definition 4. Any structure is said to be *ground* iff it contains no variables. By definition, constants are ground and variables are not ground.

Definition 5. A *term* is recursively defined as any of the following:

- a constant;
- a variable;
- a (finite) list of ground terms, denoted syntactically as `List(`t_1 t_2 ... t_n`)` where $n \geq 0$;
- a function term denoted syntactically as f`(`a_1 a_2 ... $a_{arity(f)}$`)` where f is a function name and each a_i for $1 \leq i \leq arity(f)$ is a term.

Definition 6. An *atomic formula* is any of the following:

- an *atom* denoted p`(`a_1 ... $a_{arity(p)}$`)` where p is a predicate name and each a_i for $1 \leq i \leq arity(p)$ is a term;
- an *equality formula* denoted t_1`=`t_2 where t_1 and t_2 are terms;
- a *membership formula* denoted o`#`c where o is a term called the object, and c is a term called the class;
- a *subclass formula* denoted c_1`##`c_2 where c_1 is a term called the subclass, and c_2 is a term called the superclass;
- a *frame* denoted o`[`a`->`v`]` where o is a term called the object, a is a term called the attribute, and v is a term called the value (a`->`v is referred to as the slot);[3]
- a *built-in formula*[4] denoted `External(`p`(`a_1 ... $a_{arity(p)}$`))` where p is a built-in predicate name and each a_i for $1 \leq i \leq arity(p)$ is a term.

Definition 7. A *fact* is a ground atomic formula.

Definition 8. An *independent fact* is a fact that is a built-in formula or an equality formula in which both terms are identical.

[3]Unlike RIF-PRD, I restrict frame atomic formulas to a single slot. This is just to simplify the syntax and does not reduce expressivity since the conjunction of multiple frames with the same object `And(`o`[`a_1`->`v_1`]` ... o`[`a_n`->`v_n`])` is effectively the same as a single frame having all slots o`[`a_1`->`v_1 ... a_n`->`v_n`]`.

[4]I have opted to refer to these kinds of formulas as "built-in formulas" which is the more traditional terminology. RIF-PRD refers to these as "externally-defined formulas."

Definition 9. A *dependent fact* is a fact that is not an independent fact.

The *fact* is the fundamental unit of data. The distinction between independent and dependent facts is not explicit in RIF-PRD. However, such a distinction is useful when defining the operational semantics. As will be made clearer in section 3.1.2, an independent fact is a fact that is implicitly assumed, whereas a dependent fact must be explicitly asserted.

Definition 10. A *condition* (formula) is any of the following[5]:

- an atomic formula;
- a *negated formula* denoted `Not(`f`)` where f is an atomic formula;
- a *conjunction* denoted `And(`$f_1 \ \ldots \ f_n$`)` where $n \geq 0$ and each f_i for $1 \leq i \leq n$ is either an atomic formula or a negated formula.

Notation. *For a formula f, let $\mathcal{C}^+(f)$ be defined as follows:*

- *if f is an atomic formula, then $\mathcal{C}^+(f) = \{f\}$;*
- *if f is a negated formula, then $\mathcal{C}^+(f) = \emptyset$;*
- *if f is a conjunction* `And(`$f_1 \ \ldots \ f_n$`)`*, then $\mathcal{C}^+(f) = \bigcup_{i=1}^{n} \mathcal{C}^+(f_i)$.*

Notation. *For a formula f, let $\mathcal{C}^\neg(f)$ be defined as follows:*

- *if f is an atomic formula, then $\mathcal{C}^\neg(f) = \emptyset$;*
- *if f is a negated formula* `Not(`f'`)`*, then $\mathcal{C}^\neg(f) = \{f'\}$;*
- *if f is a conjunction* `And(`$f_1 \ \ldots \ f_n$`)`*, then $\mathcal{C}^\neg(f) = \bigcup_{i=1}^{n} \mathcal{C}^\neg(f_i)$.*

Notation. *For a formula f, let $\mathcal{C}(f) = \mathcal{C}^+(f) \cup \mathcal{C}^\neg(f)$.*

[5]The following are differences with RIF-PRD: (1) existential formulas are excluded; (2) disjunction formulas are excluded; (3) the subformula of a negated formula must be atomic; (4) the subformulas of a conjunction must be either atomic or negated formulas. The only apparent reduction in expressivity is the absence of existential formulas; the other restrictions simply enforce that rules be in normal form.

Definition 11. An *action* is one of the following[6]:

- an *assertion* denoted `Assert(`f`)` where f is an atom or frame;
- a *retraction* denoted `Retract(`f`)` where f is an atom or frame.

In either case, f is referred to as the *target* of the action.

Notation. *For an action a, let $\mathcal{A}^+(a)$ be defined as follows:*

- *if a is* `Assert(`f`)`*, then $\mathcal{A}^+(a) = \{f\}$;*
- *if a is* `Retract(`f`)`*, then $\mathcal{A}^+(a) = \emptyset$.*

Notation. *For an action a, let $\mathcal{A}^\neg(a)$ be defined as follows:*

- *if a is* `Assert(`f`)`*, then $\mathcal{A}^\neg(a) = \emptyset$;*
- *if a is* `Retract(`f`)`*, then $\mathcal{A}^\neg(a) = \{f\}$.*

Notation. *For an action a, let $\mathcal{A}(a) = \mathcal{A}^+(a) \cup \mathcal{A}^\neg(a)$.*

Definition 12. An *action block* is a sequence of actions denoted `Do(`$a_1 \ \ldots \ a_n$`)` where $n \geq 1$ and each a_i for $1 \leq i \leq n$ is an action.[7]

Notation. *For an action block $\alpha =$* `Do(`$a_1 \ \ldots \ a_n$`)`*, define the following notation:*

- *for $n = 1$, $\mathcal{A}^+(\alpha) = \mathcal{A}^+(a_1)$ and $\mathcal{A}^\neg(\alpha) = \mathcal{A}^\neg(a_1)$;*
- *for $n > 1$,*
  - *if a_n is* `Assert(`f`)`*, then*
    - $\mathcal{A}^+(\alpha) = \mathcal{A}^+($`Do(`$a_1 \ \ldots \ a_{n-1}$`)`$) \cup \{f\}$,
    - $\mathcal{A}^\neg(\alpha) = \mathcal{A}^\neg($`Do(`$a_1 \ \ldots \ a_{n-1}$`)`$) \setminus \{f\}$;
  - *if a_n is* `Retract(`f`)`*, then*

[6]This definition is significantly more restrictive than in RIF-PRD. Specifically, the ability to retract all frames with a specific object and possibly a specific attribute is excluded; the `Execute` action is excluded; compound actions like the `Modify` action are excluded; and assertion of membership formulas (which requires action variable declarations) is excluded.

[7]Unlike RIF-PRD, I exclude action variable declarations. This constitutes a loss of expressivity taken to avoid the complications of introducing new constants.

  - $\mathcal{A}^+(\alpha) = \mathcal{A}^+(\texttt{Do}(a_1 \ \ldots \ a_{n-1})) \setminus \{f\}$,
  - $\mathcal{A}^\neg(\alpha) = \mathcal{A}^\neg(\texttt{Do}(a_1 \ \ldots \ a_{n-1})) \cup \{f\}$;

- $\mathcal{A}(\alpha) = \mathcal{A}^+(\alpha) \cup \mathcal{A}^\neg(\alpha)$.

It is important to note that the meaning of the notation $\mathcal{A}^+(\alpha)$ and $\mathcal{A}^\neg(\alpha)$ is carefully defined so that (if α is ground) $\mathcal{A}^+(\alpha)$ does not include asserted facts that are retracted by subsequent actions and $\mathcal{A}^\neg(\alpha)$ does not include retracted facts that are asserted by subsequent actions. This will help to avoid discussion of trivial edge cases in later proofs.

Definition 13. A rule is denoted `If` f `Then` a where f is a condition and a is an action block.[8]

Notation. *For a rule* $r =$ `If` f `Then` a*, define the following notation:*

- $\mathcal{C}^+(r) = \mathcal{C}^+(f)$, $\mathcal{C}^\neg(r) = \mathcal{C}^\neg(f)$, *and* $\mathcal{C}(r) = \mathcal{C}(f)$;
- $\mathcal{A}^+(r) = \mathcal{A}^+(a)$, $\mathcal{A}^\neg(r) = \mathcal{A}^\neg(a)$, *and* $\mathcal{A}(r) = \mathcal{A}(a)$.

Note that I have placed no significant restrictions on what constitutes a rule. For example, under this definition, it is possible that $\mathcal{C}^+(r) = \emptyset$, which immediately raises concern for the reader who is familiar with production rules. Granted, this is unusual, but there is no reason to restrict the syntax of rules here. Rather, I will address this issue in definition 24.

3.1.2 Operational Semantics

Whereas the previous section defined the syntax of rules (i.e., what rules look like), this section provides semantics for how to derive inferences and take action based on the rules. It should be understood, though, that an operational semantics differs from a declarative semantics. In a declarative semantics, terms, propositions, rules, and other structures are first given meaning, and then operations using those structures are derived which must work within the confines of those semantics. The advantage of such semantics is that meaning is objectively established. On the

[8] Unlike in RIF-PRD which uses the `Forall` keyword to indicate universal quantification of variables, herein, all variables in a rule are implicitly universally quantified and scoped to the rule.

other hand, an operational semantics gives meaning to the structures based solely on operations over the structures, and so the operations *are* the semantics. The advantage here is that it is arguably easier (for those who are not trained logicians) to understand the implications of the rules and data, even if they are not as well-rooted in a purely logic-based framework. Regardless, rules based on operational semantics are still powerful and practical, and for some forms of rules (e.g., Datalog), the declarative and operational semantics are effectively the same.

The operational semantics in this section are ultimately defined in a significantly different way than in RIF-PRD, although they are equivalent. Here the semantics are given algorithmically rather than as a labeled terminal transition system. The reformulation in this section is meant to simplify later theorems and proofs.

Definition 14. A *factset*[9] is a set of facts $F = F_I \cup F_D$ such that:

- F_I is a (possibly infinite) set of independent facts such that $f \in F_I$ iff one of the following holds,
  - f is a built-in formula that evaluates to true according to the semantics of its predicate,
  - f is an equality formula in which both sides are identical;
- F_D is a *finite* set of dependent facts satisfying the following conditions:
  - if $c_1\#\#c_2 \in F_D$ and $c_2\#\#c_3 \in F_D$, then $c_1\#\#c_3 \in F_D$;
  - if $o\#c_1 \in F_D$ and $c_1\#\#c_2 \in F_D$, then $o\#\#c_2 \in F_D$;
  - if $t_1{=}t_2 \in F_D$ and $t_2{=}t_3 \in F_D$, then $t_1{=}t_3 \in F_D$;
  - if $t_1{=}t_2 \in F_D$, then $t_2{=}t_1 \in F_D$.

A factset is the form of dataset for inference as discussed herein. The conditions in definition 14 merely ensure coherence of the factset (e.g., equality should

[9]RIF-PRD uses a similar notion referred to as a "State of the Fact Base" which effectively corresponds to F_D in the definition of factset. However, keeping independent facts "outside" of the factset causes pain in later proofs. It is easier and just as valid – or so I claim – to consider independent facts as being implicitly present in every factset.

be symmetric and transitive). These are nuances which need to be explicitly stated for completeness but which will not have much impact hereafter.

Definition 15. A *substitution* is a finite function σ from variables to terms.

Notation. *For any structure (condition, rule, etc.) s, let $\sigma(s)$ denote s having each variable v occurring in s that is also in the domain of σ, replaced by $\sigma(v)$.*

Definition 16. A *ground substitution* is a substitution such that the range consists only of ground terms.

Definition 17. A ground formula f is said to *match* a factset F iff one of the following is true:

- f is a fact and $f \in F$;
- $f = \texttt{Not}(f')$ and $f' \notin F$;
- $f = \texttt{And}(f_1 \ \ldots \ f_n)$ and for all i such that $1 \leq i \leq n$, f_i matches F.

Proposition 1. *A ground formula f matches a factset F iff the following hold:*

- $\mathcal{C}^+(f) \subseteq F$;
- $\mathcal{C}^\neg(f) \cap F = \emptyset$.

Definition 18. A non-ground formula f is said to *match* a factset F iff there exists a ground substitution σ such that $\sigma(f)$ is a ground formula that matches F.

Definition 19. The result of *applying* a ground action a to a factset F, denoted $a(F)$, is defined as follows:

- if $a = \texttt{Assert}(f)$, then $a(F) = F \cup \{f\}$;
- if $a = \texttt{Retract}(f)$, then $a(F) = F \setminus \{f\}$.

From this point on, the definitions differ significantly from RIF-PRD.

Definition 20. The result of *applying* a ground action block $\alpha = \texttt{Do}(a_1 \ \ldots \ a_n)$ to a factset F, denoted $\alpha(F)$, is defined as $\alpha(F) = a_n \circ \ldots \circ a_1(F)$.

Definition 21. The result of *firing* a ground rule $\rho = \texttt{If}\ \ f\ \ \texttt{Then}\ \ a$ on a factset F, denoted $\rho(F)$, is defined as $a(F)$ (regardless of whether f matches F).

Definition 22. A ground rule ρ is said to be an *instance* of a rule r iff there exists a ground substitution σ such that $\sigma(r) = \rho$.

Definition 23. A rule is said to be *matchable* iff there exists an instance of the rule ρ and a factset F such that the condition formula of ρ matches F. A rule that is not matchable is said to be unmatchable.

Definition 24. A rule r is said to be *finitely matchable* iff r is matchable and for any factset F, the set of all rule instances of r that match F is finite.

Definition 25. A *ruleset* is a set of finitely matchable rules.

Definition 26. Given a factset F and a ruleset R, the *conflict set* wrt R and F, denoted $conf(R, F)$, is the set of all rule instances such that for each $\rho \in conf(R, F)$, ρ is an instance of a rule in R that has a condition formula that matches F.

Proposition 2. *A ground rule ρ is in $conf(R, F)$ iff the following hold:*

- *ρ is an instance of some rule in R;*
- *$\mathcal{C}^+(\rho) \subseteq F$;*
- *$\mathcal{C}^\neg(\rho) \cap F = \emptyset$.*

Earlier in section 3.1.1, I pointed out that it was unusual that by my definition of rule, it is syntactically valid that a rule have no non-negated subformula in its condition. The peculiarity there is that it seems as though a rule could have infinitely many rule instances, and thus a conflict set could be infinite, leading to (as will be clear from later definitions) non-terminating inference cycles. However, this has been prevented by definitions 24 and 25. Although the characteristic of being "finitely matchable" is usually syntactically enforced, the nuances and complexities of the syntactic restrictions that provide such a guarantee are unnecessary for this work. All that matters here is that the rules are indeed finitely matchable.

Definition 27. *Additional information* is an intentionally vaguely defined notion of state that an inference system can keep and modify throughout the inference process, and which may affect the outcome of inference.

As stated in the definition, the idea of "additional information" is intentionally vague, and I will not make much use of it. However, it is included for compatibility purposes for the way many systems actually work. For example, in RIF-PRD, it is assumed that a history of factsets is (effectively) kept as inference progresses. Such information can be used by conflict resolution strategies and halting conditions (defined hereafter).

Definition 28. A *conflict resolution strategy* (CRS) is a function S such that for any set of matchable ground rules R, any factset F, and some additional information I, $S(I, R, F) = \langle \rho_i \rangle_{i=1}^{n}$ where $n \geq 0$, $|\{\rho_i\}_{i=1}^{n}| = n$, and $\{\rho_i\}_{i=1}^{n} \subseteq conf(R, F)$.

The definition of CRS requires some clarification. $\{\rho_i\}_{i=1}^{n} \subseteq conf(R, F)$ means that S will select a finite number of rules from the conflict set. $|\{\rho_i\}_{i=1}^{n}| = n$ means that $\langle \rho_i \rangle_{i=1}^{n}$ will not contain any duplicate ground rules.

Definition 29. A *halting condition* H is a function such that given a CRS S, a ruleset R, a factset F, and some additional information I, $H(I, S, R, F)$ is a boolean value.

Definition 30. An *information keeper* $\mathcal{I}$ is a function such that, given a halting condition H, a CRS S, a ruleset R, a factset F, and some additional information I, $\mathcal{I}(I, H, S, R, F)$ returns some additional information. For any information keeper $\mathcal{I}$, let $\mathcal{I}_\emptyset$ denote some initial, additional information.

Definition 31. A *program* is a quadruple $\langle \mathcal{I}, H, S, R \rangle$ where $\mathcal{I}$ is an information keeper, H is a halting condition, S is a CRS, and R is a ruleset.

Definition 32. A *program instance* is a quintuple $\langle \mathcal{I}, H, S, R, F \rangle$ where $\langle \mathcal{I}, H, S, R \rangle$ is a program and F is a factset.

Algorithm 1 provides the operational semantics for a single cycle in the inference process. A CRS is used to determine which rule instances in the conflict

Algorithm 1: cycle(I, S, R, F)

Data: Additional information I, a CRS S, a ruleset R, and a factset F.
Result: The factset F' resulting from a single inference cycle.

1 $\langle \rho_j \rangle_{j=1}^{n} = S(I, R, F)$
2 $F' = F$
3 **for** $j = 1$ **to** n **do**
4 $\quad F' = \rho_j(F')$
5 **end**
6 **return** F'

set are to be fired, and also *in what order* they should be fired. Then each rule instance is fired one at a time in the order determined by the CRS. Traditionally, a CRS would choose only a single rule [28]. However, the operational semantics here are fashioned after the operational semantics of RIF-PRD which allows for greater flexibility. The ability to select more than one rule instance will prove convenient for parallel inference.

Algorithm 2: infer(π)

Data: A program instance $\pi = \langle \mathcal{I}, H, S, R, F \rangle$.
Result: The *closure* F^*, if the procedure terminates.

1 $I^* = \mathcal{I}_\emptyset$
2 $F^* = F$
3 **while** *not* $H(I^*, S, R, F^*)$ **do**
4 $\quad I' = \mathcal{I}(I^*, H, S, R, F^*)$
5 $\quad F' = cycle(I^*, S, R, F^*)$
6 $\quad I^* = I'$
7 $\quad F^* = F'$
8 **end**
9 **return** F^*

Algorithm 2 provides the operational semantics for the entire inference process, which is straightforward. As long as the halting condition H does not indicate that inference should stop, additional inference cycles are performed. The result of algorithm 2 for input π is referred to as the *closure* of π.

Definition 33. The *length* ω of $infer(\pi)$ is a non-negative integer or ∞ such that ω is the number of times the loop at line 3 iterates.

Definition 34. The i^{th} *factset when calling* $infer(\pi)$ is the factset F^* from line 7 during the $\min\{\omega, i\}^{th}$ iteration of the loop at line 3, where ω is the length of $infer(\pi)$.

Note that even if the length of $infer(\pi)$ is some integer ω (not ∞), there is still a meaningful definition for the i^{th} factset when calling $infer(\pi)$ even when $i > \omega$, which is the same as the ω^{th} factset. Loosely phrased, the closure is "carried forward" indefinitely.

Definition 35. A program instance π is said to *terminate* iff the length of $infer(\pi)$ is not ∞.

Definition 36. A program Π is said to *terminate* iff every instance π of Π terminates.

3.2 Distribution

In this section, I introduce the formal concepts of parallel inference. Unlike the previous sections in this chapter that introduced variations and reformations on rules and their operational semantics as defined in RIF-PRD, this section begins the significantly novel contributions of this chapter.

Definition 37. A *distribution scheme* is a triple $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ where:

- $\mathcal{N} = \{i\}_{i=0}^{n-1}$ for some non-negative integer n;
- ϕ is a function mapping facts to subsets of $\mathcal{N}$;
- θ is a function mapping facts to *non-empty* subsets of $\mathcal{N}$;
- for any fact f,
  - $\phi(f) \subseteq \theta(f)$ and
  - $[\phi(f) = \emptyset] \rightarrow [\theta(f) = \mathcal{N}]$.

The intuition behind the definition of distribution scheme is as follows. There is some finite set of processor identifiers given by $\mathcal{N}$. For any fact f, ϕ determines

which processors *must* have f (should f be present in a factset), and θ determines which processors *may* (or are allowed to) have f. Another way to think of it is that $\phi(f)$ is the set of processors guaranteed to have f (should f be present among the processors), and $\mathcal{N} \setminus \theta(f)$ is the set of processors guaranteed *not* to have f. The necessity of ϕ is apparent, but perhaps the utility of θ is less obvious. θ is important for supporting parallel inference with negation and retraction, as will be demonstrated in later theorems and proofs. Note that there are two conditions on the relationship between ϕ and θ. The first is that, for any fact f, if a processor *must have* f, then it must also *be allowed to have* f. The second is that, if no processor is required to have f, then any processor is allowed to have f. That is, if $\mathcal{D}$ does not give any guarantee about where f is to be placed, then $\mathcal{D}$ must allow f to be placed arbitrarily.

Definition 38. Given a distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$, a *$\mathcal{D}$-distribution of a factset F* is a total function $\mathcal{F}$ from $\mathcal{N}$ to subsets of F such that the following hold:

- if $f \in F$ and $k \in \phi(f)$, then $f \in \mathcal{F}(k)$;
- if $f \in F$ and $k \notin \theta(f)$, then $f \notin \mathcal{F}(k)$;
- $F = \bigcup_{p \in \mathcal{N}} \mathcal{F}(p)$.

The $\mathcal{D}$-distribution of a factset F is an assignment of facts in F to processors identified by the integers in $\mathcal{N}$. $\mathcal{F}(k)$ is the subset of F assigned to processor k. Note that there are three conditions on a valid $\mathcal{D}$-distribution of F. First, if a fact f is in the factset F, and if k must have f, then indeed, processor k is assigned f. Second, if a fact f is in the factset F, and if k is not allowed to have f, then indeed, processor k is not assigned f. Finally, the union over the processors' assigned factsets equals the (non-distributed) factset F.

Algorithm 3 provides the operational semantics for parallel inference. It is a variation on algorithm 2 that accounts for distribution of data using the previously defined concepts. On line 1, the input factset F is distributed among processors (identified by $\mathcal{N}$) such that $\mathcal{F}$ is a $\mathcal{D}$-distribution of F. The loop at line 2 is executed in parallel as indicated by the "pardo". The body of that loop is essentially the same

Algorithm 3: parinfer($\mathcal{D}$, π)

Data: A distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ and a program instance $\pi = \langle \mathcal{I}, H, S, R, F \rangle$.

Result: The *parallel* $\mathcal{D}$*-closure* F^*, if the procedure terminates.

```
1  let F be any D-distribution of F
2  for p ∈ N pardo
3      I*_p = I_∅
4      F*_p = F(p)
5      while not H(I*_p, S, R, F*_p) do
6          I'_p = I(I*_p, H, S, R, F*_p)
7          F'_p = cycle(I*_p, S, R, F*_p)
8          I*_p = I'_p
9          F*_p = {f | f ∈ F'_p ∧ p ∈ θ(f)}
10     end
11 end
12 F* = ⋃_{p∈N} F*_p
13 return F*
```

as the contents of algorithm 2 except for one important difference at line 9. Line 9 amounts to saying that if a processor infers a fact f that it is not allowed to have (determined by θ), then f is removed directly after the cycle in which it is inferred. After each processor finishes, then on line 12, the union is taken over the processors' results, and that is the final result returned on line 13, referred to as the $\mathcal{D}$*-closure* of π.

Line 9 makes parallelization of inference slightly non-trivial. If $\theta(f) = \mathcal{N}$ for any f (i.e., any processor is allowed to have any fact), then parallelization consists of merely $\mathcal{D}$-distributing the factset and letting each processor perform sequential inference using algorithm 2. However, if there exists a fact f such that $\theta(f) \subset \mathcal{N}$ (i.e., some processor is not allowed to have some fact), then the filtering at line 9 causes algorithm 3 to be effectively different from the case in which each processor merely executes algorithm 2.

Note also that algorithm 3 does not ensure on its own that the $\mathcal{D}$-closure of π is the same as the (sequential) closure of π. It merely provides a mechanism of parallel inference. Whether the $\mathcal{D}$-closure of π is the same as the (sequential) closure of π (assuming such closures exist) depends on the relationship between $\mathcal{D}$ and π,

which is the topic of section 3.3.

Definition 39. The *length* ω_k *of* $parinfer(\mathcal{D}, \pi)$ *for processor* k is a non-negative integer or ∞ such that ω_k is the number of times in algorithm 3 that the loop at line 5 iterates when the loop at line 2 is on iteration $k \in \mathcal{N}$.

Definition 40. The i^{th} *factset for processor* p *when calling* $parinfer(\mathcal{D}, \pi)$ is the factset F_p^* from line 9 in algorithm 3 during the $\min\{\omega_p, i\}^{th}$ iteration of the loop at line 5 when the loop at line 2 is on iteration $p \in \mathcal{N}$, where ω_p is the length of $parinfer(\mathcal{D}, \pi)$ for processor p.

Put more plainly, the length of $parinfer(\mathcal{D}, \pi)$ for processor k is the number of inference cycles performed by processor k (which may be ∞). The i^{th} factset for processor k when calling $parinfer(\mathcal{D}, \pi)$ is the factset held by processor k after the i^{th} cycle, or if i is greater than the length (number of cycles) ω_k for processor k, it is the factset held by processor k after the ω_k^{th} cycle (i.e., processor k's local closure).

3.2.1 Definitions for Correct Parallel Inference

In this section, definitions are given for correctness[10] of parallel inference. Most intuitively, if the result in parallel is the same as the result in sequential, then parallel inference is considered correct.

Definition 41. Given a distribution scheme $\mathcal{D}$, a program Π is *weakly* $\mathcal{D}$*-parallel* iff for any instance π of Π, the following hold:

- $infer(\pi)$ terminates;
- $parinfer(\mathcal{D}, \pi)$ terminates;
- $infer(\pi) = parinfer(\mathcal{D}, \pi)$.

Definition 41 is the simplest definition for correct parallel inference. It only cares about the answers. It is not concerned with, for example, order of rule firings or whether the individual cycles (logically) synchronize in some way. It only cares that the final results are the same. However, not all programs are guaranteed to

[10]For logicians, correctness as defined in this section implies soundness and completeness.

terminate, and so definition 41 is not useful in those cases. Therefore, the following definition is introduced.

Definition 42. Given a distribution scheme $\mathcal{D}$, a program Π is *cyclically $\mathcal{D}$-parallel* iff for any instance π of Π, for all $i \geq 1$, letting F_i^* be the i^{th} factset when calling $infer(\pi)$, and letting $F_{p,i}^*$ be the i^{th} factset for processor p when calling $parinfer(\mathcal{D}, \pi)$, $F_i^* = \bigcup_{p \in \mathcal{N}} F_{p,i}^*$.

The intuition behind definition 42 is that the union over the processors' local factsets after i cycles should be the same as the (sequentially produced) factset after i cycles. Therefore, even if a program does not terminate, if the number of cycles is fixed, then being cyclically $\mathcal{D}$-parallel means that the results will be the same in parallel as in sequential when the input factset is $\mathcal{D}$-distributed.

Proposition 3. *If a program Π is cyclically $\mathcal{D}$-parallel and Π terminates, then Π is weakly $\mathcal{D}$-parallel.*

Proof. Straightforward from definitions 36, 41, and 42. □

Definition 42 is a strong enough definition for the goal of this work, but in traditional, parallel, production rule systems literature, a stronger notion of correct parallel inference has been used, based on the *serializability criterion* [14]. Therefore, one more definition is given for parallel inference.

Definition 43. A set of sequences $\{\langle x_{p,j} \rangle_{j=1}^{m_p}\}_{p \in \mathcal{N}}$ interleaves to a sequence $\langle y_i \rangle_{i=1}^{n}$ iff one of the following holds:

- if $n = 0$, then for all $p \in \mathcal{N}$, $m_p = 0$;
- if $n > 0$, then $\{S'_p\}_{p \in \mathcal{N}}$ interleaves to $\langle y_i \rangle_{i=2}^{n}$ where
  - there exists $p \in \mathcal{N}$ such that $m_p > 0$, $x_{p,1} = y_1$, and $S'_p = \langle x_{p,j} \rangle_{j=2}^{m_p}$,
  - for all $p \in \mathcal{N}$, either
    - $m_p > 0$, $x_{p,1} = y_1$, and $S'_p = \langle x_{p,j} \rangle_{j=2}^{m_p}$,
    - $S'_p = \langle x_{p,j} \rangle_{j=1}^{m_p}$.

Definition 44. Given a distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$, a program Π is strongly $\mathcal{D}$-parallel iff the following hold:

- Π is cyclically $\mathcal{D}$-parallel;
- for any program instance π of Π wrt to F and any $\mathcal{D}$-distribution $\mathcal{F}$ of F, letting $\langle \rho_i \rangle_{i=1}^{\omega}$ be the sequence of rule instances fired when calling $infer(\pi)$, and letting $\langle \rho_{p,j} \rangle_{j=1}^{\omega_p}$ be the sequence of rule instances fired in the $p \in \mathcal{N}$ iteration of the call to $parinfer(\mathcal{D}, \pi)$, $\{\langle \rho_{p,j} \rangle_{j=1}^{\omega_p}\}_{p \in \mathcal{N}}$ interleaves to $\langle \rho_i \rangle_{i=1}^{\omega}$.

Definition 44 is the same as definition 42 except with the additional condition that the rule instance firings of each processor in parallel inference must be able to interleave to the rule instance firings in sequential inference. Not only then are the "answers the same" (as required by being cyclically $\mathcal{D}$-parallel), but they can also be justified in a stronger sense. This notion of correct parallel inference is included for completeness with respect to previous, related work.

3.3 Sufficient Conditions

Having established definitions for rules, inference, and parallel inference, in this section, I look at sufficient conditions on distribution schemes in relation to rulesets under a certain class of CRSs and halting conditions such that, when the conditions are met, parallel inference is correct. To support these theorems, a number of additional definitions are introduced as well.

First in section 3.3.1, sufficient conditions are determined for ground rules only. Then, in section 3.3.2, the conditions from section 3.3.1 are generalized to rules (ground or otherwise). Generalizing the conditions to (general) rules is important because it allows for the conditions to be checked by direct inspection of the rules rather than considering every possible rule instance (of which there could be infinitely many).

3.3.1 Conditions on Rule Instances

First consider what it would take for a ground rule ρ that matches a factset F to match some local factset $\mathcal{F}(k)$ where $\mathcal{F}$ is a $\mathcal{D}$-distribution of F and $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$.

Clearly, some processor must have all the facts in the (non-negated) portion of the condition of ρ, that is, the facts in $\mathcal{C}^+(\rho)$. There are two ways this can happen. ϕ can guarantee that every $f \in \mathcal{C}^+(\rho)$ is assigned to some common processor, or it can guarantee that for all but at most one $f \in \mathcal{C}^+(\rho)$, f is replicated to all processors. In the latter case, the placement of the one fact $g \in \mathcal{C}^+(\rho)$ that is not replicated does not matter because whichever processor has it (and some processor will have it since $F = \bigcup_{p \in \mathcal{N}} \mathcal{F}(p)$), that processor will also have all $\mathcal{C}^+(\rho) \setminus \{g\}$. This intuition leads to the definition of a ground rule ρ being $\mathcal{D}$-matchable.

Definition 45. Let ρ be a ground rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. ρ is said to be $\mathcal{D}$-*matchable* iff one of the following holds:

- $\bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f) \neq \emptyset$;
- $\bigvee_{g \in \mathcal{C}^+(\rho)} \bigwedge_{f \in \mathcal{C}^+(\rho) \setminus \{g\}} [\phi(f) = \mathcal{N}]$.

However, it is not enough to say that if ρ matches F that it also matches some processor's local factset. It must also hold that if ρ does *not* match F, then ρ also does *not* match any processor's local factset. By proposition 1, there are two ways that a ground rule does not match a factset F. Either $\mathcal{C}^+(\rho) \nsubseteq F$ or $\mathcal{C}^\neg(\rho) \cap F \neq \emptyset$. In the former case, $\mathcal{C}^+(\rho) \nsubseteq F$ implies $\mathcal{C}^+(\rho) \nsubseteq \mathcal{F}(p)$ for all $p \in \mathcal{N}$ because $F = \bigcup_{p \in \mathcal{N}} \mathcal{F}(p)$, or put another way, for all $p \in \mathcal{N}$, $\mathcal{F}(p) \subseteq F$. Considering the latter case, that $\mathcal{C}^\neg(\rho) \cap F \neq \emptyset$, it must be ensured that for any $p \in \mathcal{N}$, if $\mathcal{C}^+(\rho) \subseteq \mathcal{F}(p)$, then $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(p) \neq \emptyset$. That is, if a processor has the facts matched by the non-negated portion of the condition of ρ, then it should also have the facts matched by the negated subformulas of ρ. Otherwise, a processor may fire ρ even though it was "blocked" from firing in sequential inference. This intuition leads to the definition of a ground rule ρ being $\mathcal{D}$-blockable.

Definition 46. Let ρ be a ground rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. ρ is said to be $\mathcal{D}$-*blockable* iff one of the following holds:

- $\mathcal{C}^\neg(\rho) = \emptyset$;
- $\bigcap_{f \in \mathcal{C}^+(\rho)} \theta(f) \subseteq \bigcap_{f \in \mathcal{C}^\neg(\rho)} \phi(f)$.

Put casually, the second condition of definition 46 says that where ever the facts in the non-negated portion of the condition of ρ *could* be placed (together), the facts in the negated portion of the condition of ρ *must* also be placed there. With definitions 45 and 46, the following two lemmas can now be formed.

Lemma 4. *Let R be a ruleset, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. If every rule instance of a rule in R is $\mathcal{D}$-matchable and $\mathcal{D}$-blockable, then for any $\mathcal{D}$-distribution $\mathcal{F}$ of a factset F, if $\rho \in conf(R, F)$ and $k \in \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$, then $\rho \in conf(R, \mathcal{F}(k))$.*

Proof. If $\rho \in conf(R, F)$, then by proposition 2, $\mathcal{C}^+(\rho) \subseteq F$ and $\mathcal{C}^\neg(\rho) \cap F = \emptyset$. If $k \in \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$, then by definition 38, $\mathcal{C}^+(\rho) \subseteq \mathcal{F}(k)$. Also, if $\mathcal{C}^\neg(\rho) \cap F = \emptyset$, by definition 38, $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(k) = \emptyset$. Therefore, by proposition 2, $\rho \in conf(R, \mathcal{F}(k))$. □

Lemma 5. *Let R be a ruleset, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. If every rule instance of a rule in R is $\mathcal{D}$-matchable and $\mathcal{D}$-blockable, then for any $\mathcal{D}$-distribution $\mathcal{F}$ of a factset F, $conf(R, F) = \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$.*

Proof. First proving that if every rule instance ρ of a rule in R is $\mathcal{D}$-matchable and $\mathcal{D}$-blockable, then $conf(R, F) \subseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$. By proposition 2, $\rho \in conf(R, F)$ iff $\mathcal{C}^+(\rho) \subseteq F$ and $\mathcal{C}^\neg(\rho) \cap F = \emptyset$. Since ρ is $\mathcal{D}$-matchable, then either

(a) $\bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f) \neq \emptyset$, or

(b) $\bigvee_{g \in \mathcal{C}^+(\rho)} \bigwedge_{f \in \mathcal{C}^+(\rho) \setminus \{g\}} [\phi(f) = \mathcal{N}]$.

By definition 38, case (a) means there exists some $k \in \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$, which by definition 38 means that $\mathcal{C}^+(\rho) \subseteq \mathcal{F}(k)$. Case (b) means that for all $p \in \mathcal{N}$, $\mathcal{C}^+(\rho) \setminus \{g\} \subseteq \mathcal{F}(p)$, and by definition 38, there is some $k \in \mathcal{N}$ such that $g \in \mathcal{F}(k)$. Therefore, whether case (a) or (b), there is some $k \in \mathcal{N}$ such that $\mathcal{C}^+(\rho) \subseteq \mathcal{F}(k)$. Additionally, by definition 38, since $\mathcal{C}^\neg(\rho) \cap F = \emptyset$, then $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(k) = \emptyset$. By proposition 2, this means that $\rho \in conf(R, \mathcal{F}(k))$, which means $\rho \in \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$. Therefore, $conf(R, F) \subseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$.

Now proving that if every rule instance ρ of a rule in R is $\mathcal{D}$-matchable and $\mathcal{D}$-blockable, then $conf(R, F) \supseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$. $\rho \in \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$

iff there exists $k \in \mathcal{N}$ such that $\rho \in conf(R, \mathcal{F}(k))$. By proposition 2, $\rho \in conf(R, \mathcal{F}(k))$ iff $\mathcal{C}^+(\rho) \subseteq \mathcal{F}(k)$ and $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(k) = \emptyset$. From definition 38, since $\mathcal{F}(k) \subseteq F$, then $\mathcal{C}^+(\rho) \subseteq F$.

For a moment, assume the possibility that $\mathcal{C}^\neg(\rho) \cap F \neq \emptyset$, in which case $\rho \notin conf(R, F)$, defeating the (real) goal of the proof. By definition 38, $k \in \bigcap_{f \in \mathcal{C}^+(\rho)} \theta(f)$, which by definition 46 implies $k \in \bigcap_{f \in \mathcal{C}^\neg(\rho)} \phi(f)$. By definition 38, this means that $\mathcal{C}^\neg(\rho) \cap F \subseteq \mathcal{F}(k)$, which means $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(k) \neq \emptyset$. However, this contradicts what has already been established, that $\mathcal{C}^\neg(\rho) \cap \mathcal{F}(k) = \emptyset$. So it is not possible that $\mathcal{C}^\neg(\rho) \cap F \neq \emptyset$.

Therefore, since $\mathcal{C}^+(\rho) \subseteq F$ and $\mathcal{C}^\neg(\rho) \cap F = \emptyset$, then by proposition 2, $\rho \in conf(R, F)$, and $conf(R, F) \supseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$.

Finally, since $conf(R, F) \subseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$ and $conf(R, F) \supseteq \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$, then $conf(R, F) = \bigcup_{p \in \mathcal{N}} conf(R, \mathcal{F}(p))$. □

Lemma 5 provides sufficient conditions for correctly matching rules (i.e., determining the conflict set) in parallel, but this is far from sufficient for correct parallel inference (of any form previously defined). The CRS of a program determines which rule instances in the conflict set are to be fired and in what order. To simplify matters, I start by considering a class of CRSs that I call AOCs, and then an even more restricted class I call RAOCs.

Definition 47. An *All-and-Ordered CRS* (AOC) is a CRS S such that S selects all the rule instances in the conflict set and orders them according to some total ordering of rule instances. An AOC effectively ignores any additional information.

Definition 48. A *Retractions-first AOC* (RAOC) is an AOC S such that for any ruleset R and factset F, for $1 \leq i \leq |S(*, R, F)|$, if the i^{th} rule instance ρ_i in $S(*, R, F)$ is such that $\mathcal{A}^\neg(\rho_i) = \emptyset$, then for $i \leq j \leq |S(*, R, F)|$, the j^{th} rule instance ρ_j in $S(*, R, F)$ is such that $\mathcal{A}^\neg(\rho_j) = \emptyset$.

An AOC may seem like an arbitrary choice in CRSs. Recall, though, that the purpose of a CRS is to provide developers with an understanding of how inference progresses [14]. Traditionally, parallel CRSs have either been non-deterministic (thus defeating the purpose of a CRS) or controlled by advanced mechanisms like

metarules [14]. Although I have specifically chosen RAOCs because they are amenable to parallelization (to be shown), AOCs in general have other good qualities that make them a sensible choice. Firstly, they are deterministic due to the total ordering on rule instances, so one can know for certain how inference will progress. Secondly, AOCs do not require any additional, complex mechanisms. Perhaps the greatest drawback, though, is that because AOCs fire the entire conflict set, the results can be non-intuitive (although not from the perspective of the *operational* semantics). For RAOCs, this is similar to using inflationary semantics in Datalog$^\neg$ inference [6], so such effects are not entirely unfamiliar.

Since an AOC effectively ignores additional information, to simplify the following, I will use a $*$ in place of additional information to indicate that any additional information can be used in that place.

In addition to restricting the class of CRSs under consideration, I also restrict the class of rules under consideration, as given in the following definitions.

Definition 49. A *polarized ground rule* is a ground rule ρ such that either $\mathcal{A}^+(\rho) = \emptyset$ or $\mathcal{A}^\neg(\rho) = \emptyset$.

Definition 50. A *polarized rule* is a rule such that every instance of the rule is polarized.

Definition 51. A *polarized ruleset* is a ruleset containing only polarized rules.

Restricting consideration to RAOCs and polarized rulesets has the advantage that in an inference cycle, all retractions will take place before any assertions, thus trivializing the issue of ordering rule instances within a single cycle. Within a single cycle, no assertion can be "undone" by a retraction, although a retraction can be "undone" by an assertion. However, any such "undoing" is part of the semantics of a RAOC, and therefore, the compatibility problem [9] is trivially resolved. Lemma 6 formally captures this characteristic and draws useful conclusions from it.

Lemma 6. *Let S be a RAOC, let R be a polarized ruleset, and let F be a factset. $f \in cycle(*, S, R, F)$ iff one of the following holds:*

- *$f \in F$ and for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^\neg(\rho)$;*

- *there exists* $\rho \in conf(R, F)$ *such that* $f \in \mathcal{A}^+(\rho)$.

Proof. First proving that if $f \in F$ and for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^\neg(\rho)$; then $f \in cycle(*, S, R, F)$. Straightforwardly, if $f \in F$ and there is no rule instance $\rho \in conf(R, F)$ that retracts f, then f will not be retracted from F when performing an inference cycle and $f \in cycle(*, S, R, F)$.

Now proving that if there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^+(\rho)$, then $f \in cycle(*, S, R, F)$. By definition 48, ρ will be fired, and since R is a polarized ruleset, $\mathcal{A}^\neg(\rho) = \emptyset$, which by definition 48 means that no rule instance will be fired after ρ that retracts facts. Therefore, once f is asserted by ρ, it will remain in the factset until the end of the cycle, and so $f \in cycle(*, S, R, F)$.

Finally proving that if $f \in cycle(*, S, R, F)$, then: $f \in F$ and for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^\neg(\rho)$; or there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^+(\rho)$. Proof by contradiction. Assume $f \in cycle(*, S, R, F)$; $f \notin F$ or there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^\neg(\rho)$; and for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^+(\rho)$. Let F' be the factset in algorithm 1 after all the retraction actions have been fired (which exists by definition 48 given that R is a polarized ruleset). Then since either $f \notin F$ or there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^\neg(\rho)$, then $f \notin F'$. By definition 48, only assertion actions remain to be fired, but since for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^+(\rho)$, then f will not be asserted, and $f \notin cycle(*, S, R, F)$. This is a contradiction. □

Corollary 7. *Let* S *be a RAOC, let* R *be a polarized ruleset, and let* F *be a factset.* $f \notin cycle(*, S, R, F)$ *iff the following hold:*

- *for all* $\rho \in conf(R, F)$, $f \notin \mathcal{A}^+(\rho)$*;*
- *if* $f \in F$*, then there exists* $\rho \in conf(R, F)$ *such that* $f \in \mathcal{A}^\neg(\rho)$.

Proof. Follows directly from lemma 6. □

In the context of RAOCs, consider parallelization of rule firing (having already addressed rule matching). Intuitively, if a fact f is retracted by a rule instance ρ, then ρ must be fired on every processor on which f might exist. This intuition leads to the following definition.

Definition 52. Let ρ be a ground rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. ρ is said to be $\mathcal{D}$-retractable iff $\bigcup_{f \in \mathcal{A}^{\neg}(\rho)} \theta(f) \subseteq \bigcap_{f \in \mathcal{C}^{+}(\rho)} \phi(f)$.

Although I refer to such rules as "$\mathcal{D}$-retractable", it is a bit of a misnomer. Whether the retraction of a fact actually succeeds depends in part on the CRS independent of $\mathcal{D}$, and so even if a ground rule ρ is $\mathcal{D}$-retractable, it is not guaranteed that the facts retracted by ρ will indeed be retracted. Regardless, the term $\mathcal{D}$-retractable is convenient for its brevity, and it should be understood (as explicitly given in the theorems) that RAOCs are the CRSs under consideration (unless stated otherwise).

Lemma 8 says that when every possible instance of a rule in R is $\mathcal{D}$-matchable, $\mathcal{D}$-blockable, and $\mathcal{D}$-retractable, then a single inference cycle will produce the same result in parallel as in sequential, when the facts are $\mathcal{D}$-distributed.

Lemma 8. *Let R be a polarized ruleset, let S be a RAOC, and let $\mathcal{D} = \langle \phi, \theta, \mathcal{N} \rangle$ be a distribution scheme. If every instance of a rule in R is $\mathcal{D}$-matchable, $\mathcal{D}$-blockable, and $\mathcal{D}$-retractable; then for any $\mathcal{D}$-distribution $\mathcal{F}$ of a factset F, $cycle(*, S, R, F) = \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$.*

Proof. First proving that $cycle(*, S, R, F) \supseteq \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$. By corollary 7, $f \notin cycle(*, S, R, F)$ implies that for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^{+}(\rho)$. By lemma 5, this also means that for all $p \in \mathcal{N}$, for all $\rho \in conf(R, \mathcal{F}(p))$, $f \notin \mathcal{A}^{+}(\rho)$.

Again, by corollary 7, $f \notin cycle(*, S, R, F)$ implies that if $f \in F$, then there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^{\neg}(\rho)$. Fix ρ accordingly. By definition 38, $f \in F$ implies that there exists $k \in \mathcal{N}$ such that $f \in \mathcal{F}(k)$. Fix k accordingly. By definition 37, this also means that $k \in \theta(f)$. Then by definition 52, $k \in \bigcap_{g \in \mathcal{C}^{+}(\rho)} \phi(f)$, and by lemma 4, $\rho \in conf(R, \mathcal{F}(k))$. k is fixed such that it is any $k \in \mathcal{N}$ such that $f \in \mathcal{F}(k)$. Therefore, it holds that for any $p \in \mathcal{N}$, if $f \in \mathcal{F}(p)$, then $\rho \in conf(R, \mathcal{F}(p))$. Additionally, at the beginning of the proof, it was established that for all $\rho' \in conf(R, \mathcal{F}(k))$, $f \notin \mathcal{A}^{+}(\rho')$. By lemma 6, this means that for all $p \in \mathcal{N}$, $f \notin cycle(*, S, R, \mathcal{F}(p))$.

Therefore $cycle(*, S, R, F) \supseteq \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$.

Now to prove that $cycle(*, S, R, F) \subseteq \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$. If $f \notin \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$, then for all $p \in \mathcal{N}$, $f \notin cycle(*, S, R, \mathcal{F}(p))$. By

corollary 7, this means that for all $p \in \mathcal{N}$, for all $\rho \in conf(R, \mathcal{F}(p))$, $f \notin \mathcal{A}^+(\rho)$, or put another way, for all $\rho \in \bigcup_{p\in\mathcal{N}} conf(R, \mathcal{F}(p))$, $f \notin \mathcal{A}^+(\rho)$. By lemma 5, this means that for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^+(\rho)$.

Suppose that for some $k \in \mathcal{N}$, $f \in \mathcal{F}(k)$. Fix k accordingly. By definition 38, $f \in F$ iff such k exists. By corollary 7, there exists some $\rho \in conf(R, \mathcal{F}(k))$ such that $f \in \mathcal{A}^\neg(\rho)$. Fix ρ accordingly. By lemma 5, since $\rho \in conf(R, \mathcal{F}(k))$, then $\rho \in conf(R, F)$. Therefore, when $f \in F$, there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^\neg(\rho)$.

So having established that for all $\rho' \in conf(R, F)$, $f \notin \mathcal{A}^+(\rho')$, and if $f \in F$, then there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^\neg(\rho)$, then by corollary 7, $f \notin cycle(*, S, R, F)$.

Therefore $cycle(*, S, R, F) \subseteq \bigcup_{p\in\mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$.

Having shown $cycle(*, S, R, F) \subseteq \bigcup_{p\in\mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$ and $cycle(*, S, R, F) \supseteq \bigcup_{p\in\mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$, it holds that $cycle(*, S, R, F) = \bigcup_{p\in\mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$. □

Now it has been shown that a single inference cycle is correct in parallel for the conditions of lemma 8. However, this is still insufficient for the entire inference process. It needs to be shown that *every* cycle is correct, not just a single cycle under certain conditions.

Consider, though, if it could be shown that after each parallel cycle, the facts are still $\mathcal{D}$-distributed. Then lemma 8 would mean that the next cycle is also correct, and the cycle after that, and so on. Enforcing θ is easy because it is built in to the parallel inference algorithm at line 9 of algorithm 3. However, it is not as straightforward to enforce ϕ. Suppose, though, that whichever processors must have an inferred fact are also some of the processors that infer that fact. Then $\mathcal{D}$-distribution would be preserved between cycles. This leads to the following definition and lemma.

Definition 53. Let ρ be a ground rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. ρ is said to be $\mathcal{D}$-preserving iff $\bigcup_{f\in\mathcal{A}^+(\rho)} \phi(f) \subseteq \bigcap_{f\in\mathcal{C}^+(\rho)} \phi(f)$.

Lemma 9. *Let R be a polarized ruleset, let S be a RAOC, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. If every instance of a rule in R is $\mathcal{D}$-matchable, $\mathcal{D}$-blockable, $\mathcal{D}$-retractable, and $\mathcal{D}$-preserving; then for any $\mathcal{D}$-distribution $\mathcal{F}$ of a factset F, $\mathcal{F}' = \{\langle p, F'_p \rangle \mid F'_p = \{f \mid f \in cycle(*, S, R, \mathcal{F}(p)) \wedge p \in \theta(f)\}\}_{p \in \mathcal{N}}$ is a $\mathcal{D}$-distribution of $cycle(*, S, R, F)$.*

Proof. By definition 38, it must be shown that:

(a) if $f \in cycle(*, S, R, F)$ and $k \in \phi(f)$, then $f \in \mathcal{F}'(k)$;

(b) if $f \in cycle(*, S, R, F)$ and $k \notin \theta(f)$, then $f \notin \mathcal{F}'(k)$;

(c) $cycle(*, S, R, F) = \bigcup_{p \in \mathcal{N}} \mathcal{F}'(p)$.

Starting with condition (a). By lemma 6, $f \in cycle(*, S, R, F)$ iff one of the following holds:

(d) $f \in F$ and for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^{\neg}(\rho)$;

(e) there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^{+}(\rho)$.

Starting with case (d), since $f \in F$ and $k \in \phi(f)$, then by definition 38, $f \in \mathcal{F}(k)$. Since for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^{\neg}(\rho)$, by lemma 5, for all $\rho \in conf(R, \mathcal{F}(k))$, $f \notin \mathcal{A}^{\neg}(\rho)$. Then by lemma 6, $f \in cycle(*, S, R, \mathcal{F}(k))$. By definition 37, since $k \in \phi(f)$, then $k \in \theta(f)$. Therefore, $f \in \mathcal{F}'(k)$.

Now turning to case (e), since there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^{+}(\rho)$ and $k \in \phi(f)$, then by definition 53, $k \in \bigcap_{g \in \mathcal{C}^{+}(\rho)} \phi(g)$. By lemma 4, this means that $\rho \in conf(R, \mathcal{F}(k))$. Then by lemma 6, $f \in cycle(*, S, R, \mathcal{F}(k))$. Since $k \in \phi(f)$, then by definition 37, $k \in \theta(f)$, and so $f \in \mathcal{F}'(k)$.

Condition (a) holds.

Condition (b) is trivially true. Regardless of whether $f \in cycle(*, S, R, F)$, by definition of $\mathcal{F}'$, if $k \notin \theta(f)$, then $f \notin \mathcal{F}'(k)$.

As for condition (c), lemma 8 already shows that $cycle(*, S, R, F) = \bigcup_{p \in \mathcal{N}} cycle(*, S, R, \mathcal{F}(p))$, and by definition of $\mathcal{F}'$, for all $p \in \mathcal{N}$, $\mathcal{F}'(p) \subseteq cycle(*, S, R, \mathcal{F}(p))$. Therefore, it holds that $\bigcup_{p \in \mathcal{N}} \mathcal{F}'(p) \subseteq cycle(*, S, R, F)$, and

it remains to be shown that $\bigcup_{p\in\mathcal{N}} \mathcal{F}'(p) \supseteq cycle(*, S, R, F)$. By lemma 6, $f \in cycle(*, S, R, F)$ iff one of case (d) or case (e) holds.

Starting with case (d), since $f \in F$, then by definition 38, there exists $k \in \mathcal{N}$ such that $f \in \mathcal{F}(k)$ and $k \in \theta(f)$. By case (d), for all $\rho \in conf(R, F)$, $f \notin \mathcal{A}^{\neg}(\rho)$, which by lemma 5 means that for all $\rho \in conf(R, \mathcal{F}(k))$, $f \notin \mathcal{A}^{\neg}(\rho)$. Therefore, by lemma 6, $f \in cycle(*, S, R, \mathcal{F}(k))$. Having already established that $k \in \theta(f)$, then $f \in \mathcal{F}'(k) \subseteq \bigcup_{p\in\mathcal{N}} \mathcal{F}'(p)$.

Turning to case (e), since there exists $\rho \in conf(R, F)$ such that $f \in \mathcal{A}^{+}(\rho)$, then by lemma 5 there exists $k \in \mathcal{N}$ such that $\rho \in conf(R, \mathcal{F}(k))$, which by lemma 6 means that $f \in cycle(*, S, R, \mathcal{F}(k))$. Now if $\phi(f) = \emptyset$, then by definition 37, $\theta(f) = \mathcal{N}$ and therefore $f \in \mathcal{F}'(k)$. If $\phi(f) \neq \emptyset$, then there exists $l \in \phi(f)$, which by definition 53 means that $l \in \bigcap_{g\in\mathcal{C}^{+}(\rho)} \phi(g)$, which by lemma 4 means that $\rho \in conf(R, \mathcal{F}(l))$, which by lemma 6 means that $f \in cycle(*, S, R, \mathcal{F}(l))$. Since it has already been established that $l \in \phi(f)$, then by definition 37, $l \in \theta(f)$, and therefore $f \in \mathcal{F}'(l)$. Either way, $f \in \bigcup_{p\in\mathcal{N}} \mathcal{F}(p)$.

Therefore, $\bigcup_{p\in\mathcal{N}} \mathcal{F}'(p) = cycle(*, S, R, F)$, and condition (c) holds.

Since (a), (b), and (c) hold, then $\mathcal{F}'$ is a $\mathcal{D}$-distribution of $cycle(*, S, R, F)$. □

Now with the support of the recently stated lemmas, the first main theorem can be concluded and proven inductively as discussed, but first, I assume a particular (kind of) halting condition.

Definition 54. A fixpoint halting condition, denoted $\mathcal{H}_{fix}$, is (logically) defined as $\mathcal{H}_{fix}(I, S, R, F) \equiv [F = cycle(I, S, R, F)]$.

Theorem 10. *Let R be a polarized ruleset, let S be a RAOC, let $\mathcal{I}$ be any information keeper, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme. If every instance of a rule in R is $\mathcal{D}$-matchable, $\mathcal{D}$-blockable, $\mathcal{D}$-retractable, and $\mathcal{D}$-preserving; then program $\Pi = \langle \mathcal{I}, \mathcal{H}_{fix}, S, R \rangle$ is cyclically $\mathcal{D}$-parallel.*

Proof. It must be shown that for any instance π of Π wrt F, for any non-negative integer i, letting F_i be the i^{th} factset when calling $infer(\pi)$ and letting $F_{p,i}$ be the i^{th} factset for processor p when calling $parinfer(\pi)$, $F_i = \bigcup_{p\in\mathcal{N}} F_{p,i}$. Proof is by induction.

Let $i = 1$ be the base case. Since $\mathcal{F}$ is a $\mathcal{D}$-distribution of F, then by lemma 9, $\{\langle p, F_{p,1}\rangle\}_{p\in\mathcal{N}}$ is a $\mathcal{D}$-distribution of F_1.

Now as the inductive step, assume that for any i, $\{\langle p, F_{p,i}\rangle\}_{p\in\mathcal{N}}$ is a $\mathcal{D}$-distribution of F_i. Then by lemma 9, $\{\langle p, F_{p,i+1}\rangle\}_{p\in\mathcal{N}}$ is a $\mathcal{D}$-distribution of F_{i+1}.

So for all $i \geq 1$, $\{\langle p, F_{p,i}\rangle\}_{p\in\mathcal{N}}$ is a $\mathcal{D}$-distribution of F_i, which by definition 38 implies that for $i \geq 1$, $F_i = \bigcup_{p\in\mathcal{N}} F_{p,i}$. □

Corollary 11. *In addition to the conditions of theorem 10, if* Π *terminates, then* Π *is weakly* $\mathcal{D}$*-parallel.*

Proof. Follows directly from theorem 10 and proposition 3. □

Turning to the possibility of inference being strongly $\mathcal{D}$-parallel, the following conjectures are given. These are presented as conjectures rather than complete theorems because their proofs are sketches. Providing complete proofs remains as future work.

Conjecture 12. *Let* R *be a ruleset, let* S *be a AOC, and let* $\mathcal{D} = \langle \mathcal{N}, \phi, \theta\rangle$ *be a distribution scheme. If every rule instance of a rule in* R *is* $\mathcal{D}$*-matchable and* $\mathcal{D}$*-blockable, then for any* $\mathcal{D}$*-distribution* $\mathcal{F}$ *of a factset* F*,* $\{S(*, R, \mathcal{F}(p))\}_{p\in\mathcal{N}}$ *interleaves to* $S(*, R, F)$*.*

Proof. (sketch) By lemma 5, $conf(R, F) = \bigcup_{p\in\mathcal{N}} conf(R, \mathcal{F}(p))$. Then, by definition 47, since S selects all the rule instances in the conflict set and orders them according to a total ordering of rule instances, then $\{S(*, R, \mathcal{F}(p))\}_{p\in\mathcal{N}}$ interleaves to $S(*, R, F)$. □

Conjecture 13. *Let* R *be a polarized ruleset, let* S *be a RAOC, let* $\mathcal{I}$ *be any information keeper, and let* $\mathcal{D} = \langle \mathcal{N}, \phi, \theta\rangle$ *be a distribution scheme. If every instance of a rule in* R *is* $\mathcal{D}$*-matchable,* $\mathcal{D}$*-blockable,* $\mathcal{D}$*-retractable, and* $\mathcal{D}$*-preserving; then program* $\Pi = \langle \mathcal{I}, \mathcal{H}_{fix}, S, R\rangle$ *is strongly* $\mathcal{D}$*-parallel.*

Proof. (sketch) By theorem 10, Π is cyclically parallel. Recall from the proof of theorem 10 (and using the same notation), for any $i \geq 1$, $\{\langle p, F_{p,i}\rangle\}_{p\in\mathcal{N}}$ is a $\mathcal{D}$-distribution of F_i. Then by conjecture 12, in every individual cycle, the sequences

of rule instances fired in parallel interleave to the sequence of rule instances fired in sequential. Therefore, chaining the parallel sequences together over the cycles and chaining the sequential sequences together over the cycles, the parallel sequences interleave to the sequential sequence. □

3.3.2 Conditions on Rules

The theorems of the previous section are only the starting points. In and of themselves, they are not very useful because the conditions are placed *ground* rules rather than (general) rules. Given a large enough domain and number of rules (which would not seem to require many), checking each individual rule instance becomes utterly impractical. Therefore in this section, I generalize the theorems of the previous section to rules. First, though, a notion of "pattern" is needed.

Definition 55. A *restriction* is a negated equality formula `Not(`v `=` t`)` where v is a non-ground term (a variable or a function term containing variables) and t is a ground term. A restriction x = `Not(`v `=` t`)` is said to restrict a formula f iff v occurs in f.

Definition 56. A *pattern* is a conjunction formula `And(`f x_1 ... x_n`)` where f is an atomic formula, $n \geq 0$, and each x_i for $1 \leq i \leq n$ is a restriction that restricts f.

The notion of a restriction is rather simple. It merely states that a variable cannot be bound to some specific value. A pattern is then just an atomic formula with an associated set of restrictions. This particular idea of "pattern" is important because it will facilitate the ability to restrict rules to improve parallelism, discussed in chapter 4.

Notation. *For any pattern P, let $\Gamma(P) = \{f \mid P$ matches $F_I \cup \{f\}\}$ where F_I is the set of all independent facts subsumed by every factset. A fact f is said to be* matched by *a pattern P iff $f \in \Gamma(P)$.*

Although briefly introduced, this particular notation is of great importance, and its full meaning should be well understood. For a pattern P, $\Gamma(P)$ represents the set of all facts that are instances of the atomic formula in P *satisfy-*

ing the associated restrictions of P. Note that although according to the operational semantics, a pattern P = `And(_a[_b->?x] Not(?x=_c))` can match a factset $\{$`_a[_b->_d]` , `_e=_f`$\} \cup F_I$, only `_a[_b->_d]` $\in \Gamma(P)$ because P can match $\{$`_a[_b->_d]`$\} \cup F_I$ but *not* $\{$`_e=_f`$\} \cup F_I$.

More notation is defined in the following, but essentially $\mathcal{X}(f)$ is the set of restrictions occurring in a formula f, and $\mathcal{P}_C^+(r)$, $\mathcal{P}_C^\neg(r)$, $\mathcal{P}_A^+(r)$, and $\mathcal{P}_A^\neg(r)$ are the sets of patterns derived from $\mathcal{C}^+(r)$, $\mathcal{C}^\neg(r)$, $\mathcal{A}^+(r)$, and $\mathcal{A}^\neg(r)$, respectively, for any rule r.

Notation. *For a condition formula* f*, let* $\mathcal{X}(f)$ *denote the set of restrictions defined as follows:*

- *if* f *is an atomic formula,* $\mathcal{X}(f) = \emptyset$*;*
- *if* f *is a negated formula that is not a restriction,* $\mathcal{X}(f) = \emptyset$*;*
- *if* f *is a restriction,* $\mathcal{X}(f) = \{f\}$*;*
- *if* f *is a conjunction* `And(`f_1 $\ldots$ f_n`)`*, then* $\mathcal{X}(f) = \bigcup_{i=1}^{n} \mathcal{X}(f_i)$.

Notation. *For a condition formula* f*,*

- $P \in \mathcal{P}_C^+(f)$ *iff* P = `And(`f' x_1 $\ldots$ x_n`)` *for some* $f' \in \mathcal{C}^+(f)$ *where* $\{x_i\}_{i=1}^n$ *is the maximum subset of* $\mathcal{X}(f)$ *such that each* x_i *restricts* f'*;*
- $P \in \mathcal{P}_C^\neg(f)$ *iff* P = `And(`f' x_1 $\ldots$ x_n`)` *for some* $f' \in \mathcal{C}^\neg(f)$ *where* $\{x_i\}_{i=1}^n$ *is the maximum subset of* $\mathcal{X}(f)$ *such that each* x_i *restricts* f'*.*

Notation. *For a rule* r = `If` f `Then` a*:*

- $\mathcal{P}_C^+(r) = \mathcal{P}_C^+(f)$*;*
- $\mathcal{P}_C^\neg(r) = \mathcal{P}_C^\neg(f)$*;*
- $P \in \mathcal{P}_A^+(r)$ *iff* P = `And(`g x_1 $\ldots$ x_m`)` *for some* $g \in \mathcal{A}^+(r)$ *and* $\{x_j\}_{j=1}^m$ *is the maximum subset of* $\mathcal{X}(f)$ *such that each* x_j *restricts* g*;*
- $P \in \mathcal{P}_A^\neg(r)$ *iff* P = `And(`g x_1 $\ldots$ x_m`)` *for some* $g \in \mathcal{A}^\neg(r)$ *and* $\{x_j\}_{j=1}^m$ *is the maximum subset of* $\mathcal{X}(f)$ *such that each* x_j *restricts* g*.*

Notation. *For a rule r, let $\Lambda(r)$ denote the set of all matchable rule instances of r.*

The following lemma proves what is rather intuitive, that the facts in a rule instance match patterns in the rule from which the rule instance is derived. However, to be thorough and to ensure consistency in the notation, proving the lemma is a necessary step.

Lemma 14. *For any rule r, the following hold:*

- $\bigcup_{\rho\in\Lambda(r)} \mathcal{C}^+(\rho) \subseteq \bigcup_{P\in\mathcal{P}_C^+(r)} \Gamma(P)$;
- $\bigcup_{\rho\in\Lambda(r)} \mathcal{C}^\neg(\rho) \subseteq \bigcup_{P\in\mathcal{P}_C^\neg(r)} \Gamma(P)$;
- $\bigcup_{\rho\in\Lambda(r)} \mathcal{A}^+(\rho) \subseteq \bigcup_{P\in\mathcal{P}_A^+(r)} \Gamma(P)$;
- $\bigcup_{\rho\in\Lambda(r)} \mathcal{A}^\neg(\rho) \subseteq \bigcup_{P\in\mathcal{P}_A^\neg(r)} \Gamma(P)$.

Proof. If $f \in \bigcup_{\rho\in\Lambda(r)} \mathcal{C}^+(\rho)$, then there exists $\rho \in \Lambda(r)$ such that $f \in \mathcal{C}^+(\rho)$. By definition 22, this means that there exists $f' \in \mathcal{C}^+(r)$ and ground substitution σ such that $f = \sigma(f')$. Since $f' \in \mathcal{C}^+(r)$, this means that there exists a pattern $P =$ `And(`f' `...)` $\in \mathcal{P}_C^+(r)$. Now if P contains any restrictions on f', then by definition of $\mathcal{P}_C^+(r)$, such restrictions must also be in $\mathcal{C}^\neg(r)$. Since ρ is matchable, then σ is such that all of the restrictions x on f' are such that $\sigma(x) =$ `Not(`t_1 `=` t_2`)` where t_1 and t_2 are different ground terms. Therefore, the ground formula $\sigma(P)$ matches $\{\sigma(f')\} \cup F_I = \{f\} \cup F_I$ (where F_I is the set of independent facts subsumed by every factset), which means that since such a σ exists, P matches $\{f\} \cup F_I$, and so $f \in \Gamma(P)$. Similar arguments for $\mathcal{C}^\neg(\rho)$, $\mathcal{A}^+(\rho)$, and $\mathcal{A}^\neg(\rho)$ with $\mathcal{P}_C^\neg(r)$, $\mathcal{P}_A^+(r)$, and $\mathcal{P}_A^\neg(r)$, respectively. □

At this point, definitions are introduced that will help to generalize away from distribution schemes that assign *facts* to processors, to pattern assignments that assign facts to processors based on which *patterns* match them. In other words, data distribution is no longer performed on a fact-by-fact basis but a pattern-by-pattern basis. This is significantly more practical because no person will likely ever decide for each individual, possible fact, to which processors the fact should be assigned and allowed. However, moving away from facts toward patterns entails additional complexity, as will be discussed as definitions are given.

Definition 57. A pattern mapper $\Phi_{\mathcal{N}}$ is a total function from patterns to subsets of $\mathcal{N} = \{i\}_{i=0}^{n-1}$ where n is some non-negative integer.

A pattern mapper is somewhat analogous to the ϕ in a distribution scheme, thus the similarity in notation. However, it maps patterns to processors instead of facts to processors.

Definition 58. A pattern assignment is a triple of pattern mappers $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ such that the following hold for any pattern P:

- $\Phi_{\mathcal{N}}(P) \subseteq \Omega_{\mathcal{N}}(P) \subseteq \Theta_{\mathcal{N}}(P)$;
- for any pattern P' such that $\Gamma(P') \subseteq \Gamma(P)$,
  - $\Phi_{\mathcal{N}}(P) \subseteq \Phi_{\mathcal{N}}(P')$,
  - $\Omega_{\mathcal{N}}(P') \subseteq \Omega_{\mathcal{N}}(P)$,
  - $\Theta_{\mathcal{N}}(P') \subseteq \Theta_{\mathcal{N}}(P)$.

A pattern assignment consists of three pattern mappers and forces coherent relationships between them. $\Theta_{\mathcal{N}}(P)$ is a set of processors that *could* be allowed to have facts in $\Gamma(P)$. $\Phi_{\mathcal{N}}(P)$ can be thought of as a set of processors where facts in $\Gamma(P)$ can *definitely* be found (if they exist in the distributed factset), whereas $\Omega_{\mathcal{N}}(P)$ is the set of processors to which facts in $\Gamma(P)$ ought to be inferred. This relationship between patterns and facts is made explicit in definition 59.

Definition 59. A distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ is said to *conform* to a pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ iff for any pattern P, the following hold:

- $\Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \Gamma(P)} \phi(f)$;
- $\bigcup_{f \in \Gamma(P)} \phi(f) \subseteq \Omega_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \Gamma(P)} \theta(f) \subseteq \Theta_{\mathcal{N}}(P)$.

When dealing with only ground rules, a distinction like that between $\Phi_{\mathcal{N}}$ and $\Omega_{\mathcal{N}}$ is unnecessary because the level of granularity is finer. With patterns, though, one can no longer say that one rule instance of r will match on processor i while

another rule instance of r will match on a different processor j because we are no longer working at that level of detail. Now it must be said that *all* the rule instances of a rule r will match on some processor(s), and *all* the rule instances of a rule r should infer to some processor(s). Thus, precision is being lost, and the conditions are moving *farther away* from being necessary (although I have not proven any of the conditions to be necessary but only sufficient). In other words, as will be shown, when the conditions for rules (to be given) are met, the conditions for rule instances are also met, but not vice versa.

As mentioned, the conditions of definition 58 enforce some basic coherence. For example, processors that are guaranteed to have all facts in $\Gamma(P)$ are also processors to which all facts in $\Gamma(P)$ must be inferred, and all processors to which all facts in $\Gamma(P)$ must be inferred are also processors that are allowed to have facts in $\Gamma(P)$. Additionally, the pattern mappers must be conscious of the relationship between patterns. This is another difference in dealing with patterns instead of directly with facts is that there are natural relationships between patterns (e.g., $\Gamma(P_1) \subseteq \Gamma(P_2)$).

The next three lemmas establish some important relationships between pattern assignments and distribution schemes that will be useful in proving the sufficient conditions for parallel inference with rules. In and of themselves, they are not particularly interesting to the overall rhetoric.

Lemma 15. *For any distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ that conforms to a pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, for any rule r, for any $\rho \in \Lambda(r)$:*

- $\bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$;
- $\bigcap_{P \in \mathcal{P}_C^-(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{C}^\neg(\rho)} \phi(f)$;
- $\bigcap_{P \in \mathcal{P}_A^+(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{A}^+(\rho)} \phi(f)$;
- $\bigcap_{P \in \mathcal{P}_A^-(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{A}^\neg(\rho)} \phi(f)$.

Proof. $k \in \bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P)$ means that for any $P \in \mathcal{P}_C^+(r)$, $k \in \Phi_{\mathcal{N}}(P)$. Then by definition 59, for any $P \in \mathcal{P}_C^+(r)$, $k \in \bigcap_{f \in \Gamma(P)} \phi(f)$. So $k \in \bigcap_{P \in \mathcal{P}_C^+(r)} \bigcap_{f \in \Gamma(P)} \phi(f)$. Now consider the f over which intersection is occurring. It is for all $f \in \bigcup_{P \in \mathcal{P}_C^+(r)} \Gamma(P)$. By lemma 14, $\bigcup_{\rho \in \Lambda(r)} \mathcal{C}^+(\rho) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Gamma(P)$. Therefore,

intersecting over $f \in \bigcup_{\rho \in \Lambda(r)} \mathcal{C}^+(\rho)$ will be no more restrictive and produce a superset. Hence, $\bigcap_{P \in \mathcal{P}_C^+(r)} \bigcap_{f \in \Gamma(P)} \phi(f) \subseteq \bigcap_{\rho \in \Lambda(r)} \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$. Then for any $\rho^* \in \Lambda(r)$, it trivially holds that $\bigcap_{\rho \in \Lambda(r)} \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f) \subseteq \bigcap_{f \in \mathcal{C}^+(\rho^*)} \phi(f)$. By transitivity, for any $\rho^* \in \Lambda(r)$, $\bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{C}^+(\rho^*)} \phi(f)$. Similar arguments for $\mathcal{C}^\neg(\rho)$, $\mathcal{A}^+(\rho)$, and $\mathcal{A}^\neg(\rho)$ with $\mathcal{P}_C^\neg(r)$, $\mathcal{P}_A^+(r)$, and $\mathcal{P}_A^\neg(r)$, respectively. □

Lemma 16. *For any distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ that conforms to a pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, for any rule r, for any $\rho \in \Lambda(r)$:*

- $\bigcup_{f \in \mathcal{C}^+(\rho)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Omega_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{C}^\neg(\rho)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_C^\neg(r)} \Omega_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{A}^+(\rho)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_A^+(r)} \Omega_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{A}^\neg(\rho)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_A^\neg(r)} \Omega_{\mathcal{N}}(P)$.

Proof. For any rule instance $\rho^* \in \Lambda(r)$, it holds that $\bigcup_{f \in \mathcal{C}^+(\rho^*)} \phi(f) \subseteq \bigcup_{\rho \in \Lambda(r)} \bigcup_{f \in \mathcal{C}^+(\rho)} \phi(f)$. Now consider the f over which union is occurring. It is for all $f \in \bigcup_{\rho \in \Lambda(r)} \mathcal{C}^+(\rho)$. By lemma 14, $\bigcup_{\rho \in \Lambda(r)} \mathcal{C}^+(\rho) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Gamma(P)$. Therefore, union over $f \in \bigcup_{P \in \mathcal{P}_C^+(r)} \Gamma(P)$ will be no less inclusive and produce a superset. Hence, $\bigcup_{\rho \in \Lambda(r)} \bigcup_{f \in \mathcal{C}^+(\rho)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \bigcup_{f \in \Gamma(P)} \phi(f)$, and by definition 59, $\bigcup_{P \in \mathcal{P}_C^+(r)} \bigcup_{f \in \Gamma(P)} \phi(f) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Omega_{\mathcal{N}}(P)$. Similar arguments for $\mathcal{C}^\neg(\rho)$, $\mathcal{A}^+(\rho)$, and $\mathcal{A}^\neg(\rho)$ with $\mathcal{P}_C^\neg(r)$, $\mathcal{P}_A^+(r)$, and $\mathcal{P}_A^\neg(r)$, respectively. □

Lemma 17. *For any distribution scheme $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ that conforms to a pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, for any rule r, for any $\rho \in \Lambda(r)$:*

- $\bigcup_{f \in \mathcal{C}^+(\rho)} \theta(f) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Theta_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{C}^\neg(\rho)} \theta(f) \subseteq \bigcup_{P \in \mathcal{P}_C^\neg(r)} \Theta_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{A}^+(\rho)} \theta(f) \subseteq \bigcup_{P \in \mathcal{P}_A^+(r)} \Theta_{\mathcal{N}}(P)$;
- $\bigcup_{f \in \mathcal{A}^\neg(\rho)} \theta(f) \subseteq \bigcup_{P \in \mathcal{P}_A^\neg(r)} \Theta_{\mathcal{N}}(P)$.

Proof. Similar arguments as for lemma 16. □

The following four lemmas give the sufficient conditions for correct parallel inference with rules. The goal here is to determine conditions on the rules such that something can be said about all the rule instances of those rules and then tie the conclusions into previously stated theorems about rule instances. The first lemma proves conditions on a rule that are sufficient for showing that all the rule instances of that rule are $\mathcal{D}$-matchable. The second, third, and fourth lemmas show the same for $\mathcal{D}$-blockable, $\mathcal{D}$-retractable, and $\mathcal{D}$-preserving, respectively.

Lemma 18. *Let r be a rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme that conforms to pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$. If one of the following holds:*

- $\bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) \neq \emptyset$, *or*
- $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} [\Phi_{\mathcal{N}}(P) = \mathcal{N}]$;

then every $\rho \in \Lambda(r)$ is $\mathcal{D}$-matchable.

Proof. If $\bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) \neq \emptyset$, then by lemma 15, for any $\rho \in \Lambda(r)$, since $\emptyset \subset \bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f)$, then $\bigcap_{f \in \mathcal{C}^+(\rho)} \phi(f) \neq \emptyset$.

If $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \Phi_{\mathcal{N}}(P) = \mathcal{N}$, then for some $Q \in \mathcal{P}_C^+(r)$, $\bigcap_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \Phi_{\mathcal{N}}(P) = \mathcal{N}$. By definition of $\mathcal{P}_C^+(r)$, this means that for any $\rho \in \Lambda(r)$, there exists at most one $g \in \mathcal{C}^+(\rho)$ such that $\phi(g) \neq \mathcal{N}$, which means $\bigwedge_{f \in \mathcal{C}^+(\rho) \setminus \{g\}} [\phi(f) = \mathcal{N}]$. Therefore, for any $\rho \in \Lambda(r)$, the conditions of definition 45 hold true, and ρ is $\mathcal{D}$-matchable. □

Lemma 19. *Let r be a rule, and let $\mathcal{D} = \langle \mathcal{N}, \phi, \theta \rangle$ be a distribution scheme that conforms to pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$. If one of the following holds:*

- $\mathcal{P}_C^\neg(r) = \emptyset$, *or*
- $\bigcup_{P \in \mathcal{P}_C^+(r)} \Theta_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^\neg(r)} \Phi_{\mathcal{N}}(P)$;

then every $\rho \in \Lambda(r)$ is $\mathcal{D}$-blockable.

Proof. If $\mathcal{P}_C^\neg(r) = \emptyset$, then for any $\rho \in \Lambda(r)$, $\mathcal{C}^\neg(\rho) = \emptyset$.

If $\bigcup_{P \in \mathcal{P}_C^+(r)} \Theta_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^\neg(r)} \Phi_{\mathcal{N}}(P)$, then for any $\rho \in \Lambda(r)$, lemma 17 implies that $\bigcup_{f \in \mathcal{C}^+(\rho)} \theta(f) \subseteq \bigcup_{P \in \mathcal{P}_C^+(r)} \Theta_{\mathcal{N}}(P)$, and lemma 15 implies that

$\bigcap_{P\in\mathcal{P}_C^-(r)}\Phi_\mathcal{N}(P)\subseteq\bigcap_{f\in\mathcal{C}^\neg(\rho)}\phi(f)$. By transitivity, $\bigcup_{f\in\mathcal{C}^+(\rho)}\theta(f)\subseteq\bigcap_{f\in\mathcal{C}^\neg(\rho)}\phi(f)$. Note that $\bigcap_{f\in\mathcal{C}^+(\rho)}\theta(f)\subseteq\bigcup_{f\in\mathcal{C}^+(\rho)}\theta(f)$, so it further holds that $\bigcap_{f\in\mathcal{C}^+(\rho)}\theta(f)\subseteq\bigcap_{f\in\mathcal{C}^\neg(\rho)}\phi(f)$ for any $\rho\in\Lambda(r)$ Therefore, for any $\rho\in\Lambda(r)$, the conditions of definition 46 hold true, and ρ is $\mathcal{D}$-blockable. □

Lemma 20. *Let r be a rule, and let $\mathcal{D}=\langle\mathcal{N},\phi,\theta\rangle$ be a distribution scheme that conforms to pattern assignment $\langle\Phi_\mathcal{N},\Omega_\mathcal{N},\Theta_\mathcal{N}\rangle$ which cover the condition formula of r. If $\bigcup_{P\in\mathcal{P}_A^-(r)}\Theta_\mathcal{N}(P)\subseteq\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)$, then every $\rho\in\Lambda(r)$ is $\mathcal{D}$-retractable.*

Proof. If $\bigcup_{P\in\mathcal{P}_A^-(r)}\Theta_\mathcal{N}(P)\subseteq\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)$, then by lemma 17, $\bigcup_{f\in\mathcal{A}^\neg(\rho)}\theta(f)\subseteq\bigcup_{P\in\mathcal{P}_A^-(r)}\Theta_\mathcal{N}(P)$, and by lemma 15, $\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)\subseteq\bigcap_{f\in\mathcal{C}^+(\rho)}\phi(f)$. By transitivity, for any $\rho\in\Lambda(r)$, $\bigcup_{f\in\mathcal{A}^\neg(\rho)}\theta(f)\subseteq\bigcap_{f\in\mathcal{C}^+(\rho)}\phi(f)$. Therefore, for any $\rho\in\Lambda(r)$, the condition of definition 52 holds true, and ρ is $\mathcal{D}$-retractable. □

Lemma 21. *Let r be a rule, and let $\mathcal{D}=\langle\mathcal{N},\phi,\theta\rangle$ be a distribution scheme that conforms to pattern assignments $\langle\Phi_\mathcal{N},\Omega_\mathcal{N},\Theta_\mathcal{N}\rangle$. If $\bigcup_{P\in\mathcal{P}_A^+(r)}\Omega_\mathcal{N}(P)\subseteq\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)$, then every $\rho\in\Lambda(r)$ is $\mathcal{D}$-preserving.*

Proof. If $\bigcup_{P\in\mathcal{P}_A^+(r)}\Omega_\mathcal{N}(P)\subseteq\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)$, then by lemma 16, $\bigcup_{f\in\mathcal{A}^+(\rho)}\phi(f)\subseteq\bigcup_{P\in\mathcal{P}_A^+(r)}\Omega_\mathcal{N}(P)$, and by lemma 15, $\bigcap_{P\in\mathcal{P}_C^+(r)}\Phi_\mathcal{N}(P)\subseteq\bigcap_{f\in\mathcal{C}^+(\rho)}\phi(f)$. By transitivity, for any $\rho\in\Lambda(r)$, $\bigcup_{f\in\mathcal{A}^+(\rho)}\phi(f)\subseteq\bigcap_{f\in\mathcal{C}^+(\rho)}\phi(f)$. Therefore, for any $\rho\in\Lambda(r)$, the condition of definition 53 holds true, and ρ is $\mathcal{D}$-preserving. □

Finally, it can be said that sufficient conditions on rules have been proven for correct parallel inference, and this section ends with that very corollary.

Corollary 22. *Let R be a polarized ruleset, let S be a RAOC, let $\mathcal{I}$ be any information keeper, and let $\langle\Phi_\mathcal{N},\Omega_\mathcal{N},\Theta_\mathcal{N}\rangle$ be a pattern assignment. If the conditions of lemmas 18, 19, 20, and 21 are met for every rule $r\in R$, then program $\Pi=\langle\mathcal{I},\mathcal{H}_{fix},S,R\rangle$ is cyclically $\mathcal{D}$-parallel where $\mathcal{D}$ is any distribution scheme that conforms to $\langle\Phi_\mathcal{N},\Omega_\mathcal{N},\Theta_\mathcal{N}\rangle$.*

Proof. Follows immediately from lemmas 18, 19, 20, and 21; and theorem 10. □

3.4 Summary

This chapter defined the syntax and mathematical notation for rules in section 3.1.1 and defined an operational semantics in section 3.1.2. In section 3.2, definitions and operational semantics were given for *parallel* inference, and section 3.2.1 provided definitions for what is means for parallel inference to be correct. Definition 42 is the definition for correctness relied upon for the remainder of this thesis.

Section 3.3 contained the significantly novel contribution of this chapter. In section 3.3.1, sufficient conditions are determined for ground rules such that, when the conditions are met for every possible ground rule in inference, parallel inference is guaranteed to be correct relative to a distribution scheme. To be more useful, though, the conditions are generalized to rules (ground or otherwise) in section 3.3.2. These conditions provide the foundation for the findings of the following chapter.

CHAPTER 4
PRACTICAL APPLICATION OF CONDITIONS FOR PARALLEL INFERENCE

In this chapter, the sufficient conditions determined in the previous chapter are used to derive a method to restrict (polarized) rulesets such that parallel inference with the restricted version of the ruleset (with a RAOC) is correct. In section 4.1, a special class of distribution schemes – called replication schemes – is considered, and new, simpler notation is defined. The sufficient conditions from the previous chapter are then recast into the simpler notation, which reveals a possible reduction to satisfiability. This possibility is further explored and confirmed in section 4.2. Specifically, testing sufficient conditions for correct parallel inference with a replication scheme is reducible to 2SAT. Then, the 2SAT reduction is augmented to a 3SAT reduction that allows for the option to sacrifice rules in order to improve parallel inference. The problem arises, then, that the search space for solutions to the 3SAT formula becomes quite large for even moderately sized rulesets. Therefore, a methodology is proposed for reducing the search space in section 4.2.3, and it is applied to restrict the RDFS and OWL2RL rulesets in section 4.3.

4.1 Replication Schemes

In this section, a specific class of distribution schemes is introduced called replication schemes. General distribution schemes are difficult to manage because they require that, for every processor, it must be decided whether it is allowed to have a given fact, and if so, whether it *must* have that fact. Although pattern assignments allow assignment of facts by looking at a finite set of patterns, this actually complicates the process, even though it makes it more tractable. That is, not every possible fact needs to be considered (of which there could be infinitely many), but for each processor, it must be decided whether the processor is allowed to have facts matched by the pattern; and if so, whether inferences matched by the pattern must go to that processor; and if so, whether it should be guaranteed that

the processor has such facts. Additionally, the pattern assignment must be checked for validity as given by the conditions in definition 58.

A simpler form of distribution scheme which has previously been useful in [20, 21] is to restrict fact assignment to two possibilities: for any fact, either replicate it to all processors, or place it arbitrarily to some processor(s). Definition 60 captures this notion. Note that even though it is possible that for a fact f, $\phi(f) = \emptyset$, it must still hold that f is placed at *some* processor when distributed because the union across the processors' factsets must equal the factset prior to distribution (by definition 38).

Definition 60. A *replication scheme* is a distribution scheme $\mathcal{R} = \langle \mathcal{N}, \phi, \theta \rangle$ such that for any fact f:

- $\phi(f) = \emptyset$ or $\phi(f) = \mathcal{N}$;
- $\theta(f) = \mathcal{N}$.

Definition 61. A *pattern replicator* is a pattern assignment $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ such that for any pattern P:

- $\Phi_{\mathcal{N}}(P) = \emptyset$ or $\Phi_{\mathcal{N}}(P) = \mathcal{N}$;
- $\Omega_{\mathcal{N}}(P) = \emptyset$ or $\Omega_{\mathcal{N}}(P) = \mathcal{N}$;
- $\Theta_{\mathcal{N}}(P) = \mathcal{N}$.

At this point, it is useful to recast previous theorems and definitions in terms of replication schemes, and these new theorems and definitions will provide the basis for the main findings with regard to replication schemes and pattern replicators. Lemma 23 builds on definition 59 to prove what it means for a replication scheme to conform to a pattern replicator.

The gist of lemma 23 is as follows. If some processor must guarantee that it has facts matched by a pattern, then that guarantee is made for all processors. If inferences matched by a pattern need not be inferred to any particular processor, then no guarantee is made about the particular placement of such facts. Also, facts matched by any pattern are allowed to be placed at any processor. These are proven to hold true for any replication scheme conforming to a pattern replicator.

Lemma 23. *A replication scheme $\mathcal{R} = \langle \mathcal{N}, \phi, \theta \rangle$ conforms to a pattern replicator $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ iff the following hold for any pattern P:*

- *if $\Phi_{\mathcal{N}}(P) = \mathcal{N}$, then for all $f \in \Gamma(P)$, $\phi(f) = \mathcal{N}$;*
- *if $\Omega_{\mathcal{N}}(P) = \emptyset$, then for all $f \in \Gamma(P)$, $\phi(f) = \emptyset$;*
- $\Theta_{\mathcal{N}}(P) = \mathcal{N}$.

Proof. ($\rightarrow$) Assume that $\mathcal{R}$ conforms to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$.

By definition 59, it must hold for any pattern P that $\Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \Gamma(P)} \phi(f)$. If $\Phi_{\mathcal{N}}(P) = \mathcal{N}$, then $\bigcap_{f \in \Gamma(P)} \phi(f) = \mathcal{N}$, which means that for all $f \in \Gamma(P)$, $\phi(f) = \mathcal{N}$.

By definition 59, it must hold for any pattern P that $\bigcup_{f \in \Gamma(P)} \phi(f) \subseteq \Omega_{\mathcal{N}}(P)$. For any pattern P, if $\Omega_{\mathcal{N}}(P) = \emptyset$, then $\bigcup_{f \in \Gamma(P)} \phi(f) = \emptyset$, which means that for all $f \in \Gamma(P)$, $\phi(f) = \emptyset$.

By definition 59, it must hold for any pattern P that $\bigcup_{f \in \Gamma(P)} \theta(f) \subseteq \Theta_{\mathcal{N}}(P)$. By definition 60, for any fact f, $\theta(f) = \mathcal{N}$. This means that $\mathcal{N} = \bigcup_{f \in \Gamma(P)} \theta(f) \subseteq \Theta_{\mathcal{N}}(P)$.

($\leftarrow$) Proof by contradiction. Assume that the three bulleted conditions hold true but $\mathcal{R}$ does not conform to pattern replicator $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$.

If $\Phi_{\mathcal{N}}(P) = \emptyset$, then it is trivially true that $\Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \Gamma(P)} \phi(f)$. If $\Phi_{\mathcal{N}}(P) = \mathcal{N}$, then $\bigcap_{f \in \Gamma(P)} \phi(f) = \mathcal{N}$, and so it is again trivially true that $\Phi_{\mathcal{N}}(P) \subseteq \bigcap_{f \in \Gamma(P)} \phi(f)$. Therefore, if $\mathcal{R}$ does not conform to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, it is not because $\Phi_{\mathcal{N}}(P) \nsubseteq \bigcap_{f \in \Gamma(P)} \phi(f)$.

If $\Omega_{\mathcal{N}}(P) = \mathcal{N}$, this it is trivially true that $\bigcup_{f \in \Gamma(P)} \phi(f) \subseteq \Omega_{\mathcal{N}}(P)$. If $\Omega_{\mathcal{N}}(P) = \emptyset$, then $\bigcup_{f \in \Gamma(P)} \phi(f) = \emptyset$, and again, it is trivially true that $\bigcup_{f \in \Gamma(P)} \phi(f) \subseteq \Omega_{\mathcal{N}}(P)$. Therefore, if $\mathcal{R}$ does not conform to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, it is not because $\bigcup_{f \in \Gamma(P)} \phi(f) \nsubseteq \Omega_{\mathcal{N}}(P)$.

Since $\Theta_{\mathcal{N}}(P) = \mathcal{N}$, it is trivially true that $\bigcup_{f \in \Gamma(P)} \theta(f) \subseteq \Theta_{\mathcal{N}}(P)$. Therefore, if $\mathcal{R}$ does not conform to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, it is not because $\bigcup_{f \in \Gamma(P)} \theta(f) \nsubseteq \Theta_{\mathcal{N}}(P)$.

Therefore, it cannot be that $\mathcal{R}$ does not conform to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$, which is a contradiction. □

Notation. *Now that the choices are either between $\emptyset$ and $\mathcal{N}$, there is no longer a*

need for the complexity in notation brought by working with subsets of $\mathcal{N}$*. For a replication scheme* $\mathcal{R} = \langle \mathcal{N}, \phi, \theta \rangle$*, let* $\mathcal{R}(f) \equiv [\phi(f) = \mathcal{N}]$ *and* $\neg\mathcal{R}(f) \equiv [\phi(f) = \emptyset]$*.*

Let a pattern replicator be denoted as a pair $\langle \epsilon, \alpha \rangle$ *where* $\epsilon(P) \equiv [\Omega_{\mathcal{N}}(P) = \emptyset]$*,* $\neg\epsilon(P) \equiv [\Omega_{\mathcal{N}}(P) = \mathcal{N}]$*,* $\alpha(P) \equiv [\Phi_{\mathcal{N}}(P) = \mathcal{N}]$*, and* $\neg\alpha(P) \equiv [\Phi_{\mathcal{N}}(P) = \emptyset]$*.*

Corollary 24 is a direct recasting of lemma 23, illustrating the neatness of the new notation and justifying the intuition of the notation. The formulas in corollary 24 seem self-justifying. That is, $\alpha(P)$ implies that all the facts in $\Gamma(P)$ are replicated to all processors, and $\epsilon(P)$ implies that all the facts in $\Gamma(P)$ are placed arbitrarily. Note, though, that it is possible that for a pattern P, $\neg\alpha(P) \wedge \neg\epsilon(P)$. That is, patterns do not have to fall into one of these two categories, although intuitively, a pattern cannot be in both categories.

Corollary 24. *A replication scheme* $\mathcal{R}$ *conforms to pattern replicator* $\langle \epsilon, \alpha \rangle$ *iff the following hold for any pattern* P*:*

- $\alpha(P) \rightarrow \bigwedge_{f \in \Gamma(P)} \mathcal{R}(f)$*;*
- $\epsilon(P) \rightarrow \bigwedge_{f \in \Gamma(P)} \neg\mathcal{R}(f)$*.*

Proof. Straightforward rewriting of lemma 23 using the new notation. □

Finally, the previous lemma and corollary can be used to prove sufficient conditions for parallel inference with replication schemes (with polarized rulesets and RAOCs) following quickly from lemmas 18, 19, 20, and 21. To briefly summarize the corollaries, let $\mathcal{R}$ be a replication scheme that conforms to pattern assignment $\langle \epsilon, \alpha \rangle$. Corollary 25 says that if all but at most one pattern in a rule condition have their facts replicated, then all the instances of the rule are $\mathcal{R}$-matchable. Corollary 26 states that if all facts matched by patterns corresponding to negated formulas in a rule have their facts replicated, then every instance of that rule is $\mathcal{R}$-blockable. Corollary 27 says that if a rule with retract actions has all the facts matching patterns in its condition replicated, then every instance of the rule is $\mathcal{R}$-retractable. (A rule without retract actions is inherently $\mathcal{R}$-retractable.) Corollary 28 says that if a rule is such that either all the facts it can infer can be placed arbitrarily *or* all the facts matched by its condition are replicated, then every instance of the rule is $\mathcal{R}$-preserving.

Corollary 25. *Let r be a rule, and let $\mathcal{R}$ be a replication scheme that conforms to pattern replicator $\langle \epsilon, \alpha \rangle$. If $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)$, then every $\rho \in \Lambda(r)$ is $\mathcal{R}$-matchable.*

Proof. Let $\langle \epsilon, \alpha \rangle$ correspond to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ as described in the notation. If $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)$, then $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \Phi_{\mathcal{N}}(P) = \mathcal{N}$, which by lemma 18 means that every $\rho \in \Lambda(r)$ is $\mathcal{R}$-matchable. □

Corollary 26. *Let r be a rule, and let $\mathcal{R}$ be a replication scheme that conforms to pattern replicator $\langle \epsilon, \alpha \rangle$. If $\bigwedge_{P \in \mathcal{P}_C^-(r)} \alpha(P)$, then every $\rho \in \Lambda(r)$ is $\mathcal{R}$-blockable.*

Proof. Let $\langle \epsilon, \alpha \rangle$ correspond to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ as described in the notation. If $\bigwedge_{P \in \mathcal{P}_C^-(r)} \alpha(P)$, then $\bigwedge_{P \in \mathcal{P}_C^-(r)} \Phi_{\mathcal{N}}(P) = \mathcal{N}$. Recall from lemma 23 that for any pattern P, $\Theta_{\mathcal{N}}(P) = \mathcal{N}$. This means that $\bigcup_{P \in \mathcal{P}_C^+(r)} \Theta_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^-(r)} \Phi_{\mathcal{N}}(P)$, which by lemma 19 means that every $\rho \in \Lambda(r)$ is $\mathcal{R}$-blockable. □

Corollary 27. *Let r be a rule, and let $\mathcal{R}$ be a replication scheme that conforms to pattern replicator $\langle \epsilon, \alpha \rangle$. If $\mathcal{P}_A^-(r) = \emptyset$ or $\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)$, then every $\rho \in \Lambda(r)$ is $\mathcal{R}$-retractable.*

Proof. Let $\langle \epsilon, \alpha \rangle$ correspond to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ as described in the notation. If $\mathcal{P}_A^-(r) = \emptyset$, then it is trivially true that $\bigcup_{P \in \mathcal{P}_A^-(r)} \Theta_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P)$ since $\bigcup_{P \in \emptyset} \Theta_{\mathcal{N}}(P) = \emptyset$. If $\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)$, then $\bigwedge_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) = \mathcal{N}$. Recall from lemma 23 that for any pattern P, $\Theta_{\mathcal{N}}(P) = \mathcal{N}$. This means that $\bigcup_{P \in \mathcal{P}_A^-(r)} \Theta_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P)$, which by lemma 20 means that every $\rho \in \Lambda(r)$ is $\mathcal{R}$-retractable. □

Corollary 28. *Let r be a rule, and let $\mathcal{R}$ be a replication scheme that conforms to pattern replicator $\langle \epsilon, \alpha \rangle$. If $[\bigvee_{P \in \mathcal{P}_A^+(r)} \neg\epsilon(P)] \rightarrow [\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)]$, then every $\rho \in \Lambda(r)$ is $\mathcal{R}$-preserving.*

Proof. Let $\langle \epsilon, \alpha \rangle$ correspond to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ as described in the notation. If $[\bigvee_{P \in \mathcal{P}_A^+(r)} \neg\epsilon(P)] \rightarrow [\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)]$, then $[\bigvee_{P \in \mathcal{P}_A^+(r)} \Omega_{\mathcal{N}}(P) = \mathcal{N}] \rightarrow [\bigwedge_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) = \mathcal{N}]$, which is equivalently stated that $[\bigcup_{P \in \mathcal{P}_A^+(r)} \Omega_{\mathcal{N}}(P) = \mathcal{N}] \rightarrow [\bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P) = \mathcal{N}]$, which also means (taking into consideration definition 61)

$\bigcup_{P \in \mathcal{P}_A^+(r)} \Omega_{\mathcal{N}}(P) \subseteq \bigcap_{P \in \mathcal{P}_C^+(r)} \Phi_{\mathcal{N}}(P)$. Then by lemma 21, every $\rho \in \Lambda(r)$ is $\mathcal{R}$-preserving. □

These conditions, particularly those placed on negation and retraction, are quite restrictive. Starting with sufficient conditions on rule instances, then sufficient conditions on rules, and now sufficient conditions on rules for replication schemes, the conditions are becoming increasingly restrictive and unnecessary (unnecessary in the sense of "necessary and sufficient conditions"). However, as imprecision of the conditions increases, it appears that their simplicity and utility also increases, as is demonstrated in the following section.

4.2 Reductions to Satisfiability

By inspection, the conditions of the previous four corollaries are clearly satisfiability formulas, which implies that checking the conditions can be done by reduction to satisfiability. Specifically, as shown in section 4.2.1, they can be checked by reduction to 2SAT. 2SAT is a particularly desirable version of the SAT problem because it can be solved (efficiently) in polynomial time.

While merely checking the conditions for parallel inference is useful, many non-trivial rulesets will likely require all data to be replicated. Thus, it would be useful to determine which rules (or restricted versions thereof) can be eliminated to increase parallelism. In section 4.2.2, it is shown that augmenting the 2SAT reduction to a 3SAT reduction allows for the possibility to consider elimination of rules. As a result, though, the search space for 3SAT solutions can become insurmountable, and so in section 4.2.3, a methodology is proposed for reducing the search space so that a restricted version of a ruleset for parallel inference can quickly be converged upon. This methodology is then used in section 4.3 to restrict the RDFS and OWL2RL rulesets into rulesets that are amenable to parallel inference.

4.2.1 Checking Conditions by Reduction to 2SAT

In this section, it is shown how checking the sufficient conditions for replication schemes can be reduced to 2SAT. First, the problem must be clearly defined in terms of input and output.

Problem 1 (RConds)**.** Given a ruleset R and a pattern replicator $\langle \epsilon, \alpha \rangle$, determine whether the conditions of corollaries 25, 26, 27, and 28 are satisfied.

Lemma 29 states that not only are the conditions of the previous four corollaries SAT formulas, but they can also be equivalently written as 2SAT formulas.

Lemma 29. *RConds is reducible to 2SAT.*

Proof. Consider the condition of corollary 25, that $\bigvee_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)$. Suppose that for some $Q \in \mathcal{P}_C^+(r)$, $\neg\alpha(Q)$. Then the previous formula is true iff $\bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)$. In other words, the formula can be equivalently formulated as $\bigwedge_{Q \in \mathcal{P}_C^+(r)} [\neg\alpha(Q) \rightarrow \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)]$, which can be further reformulated as $\bigwedge_{Q \in \mathcal{P}_C^+(r)} [\alpha(Q) \vee \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} \alpha(P)]$, and then performing distribution, $\bigwedge_{Q \in \mathcal{P}_C^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r) \setminus \{Q\}} [\alpha(Q) \vee \alpha(P)]$. This is a 2SAT formula.

Consider the condition of corollary 26. $\bigwedge_{P \in \mathcal{P}_C^-(r)} \alpha(P)$ can be equivalently reformulated $\bigwedge_{P \in \mathcal{P}_C^-(r)} \alpha(P) \vee \alpha(P)$ which is a 2SAT formula.

Consider the condition of corollary 27. $\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)$ can be equivalently reformulated $\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P) \vee \alpha(P)$ which is a 2SAT formula.

Consider the condition of corollary 28. $[\bigvee_{Q \in \mathcal{P}_A^+(r)} \neg\epsilon(Q)] \rightarrow [\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)]$ can be equivalently reformulated as $[\bigwedge_{Q \in \mathcal{P}_A^+(r)} \epsilon(Q)] \vee [\bigwedge_{P \in \mathcal{P}_C^+(r)} \alpha(P)]$, which is equivalent to $\bigwedge_{Q \in \mathcal{P}_A^+(r)} \bigwedge_{P \in \mathcal{P}_C^+(r)} \epsilon(Q) \vee \alpha(P)$, which is a 2SAT formula.

Taking the conjunction of all these 2SAT formulas forms a larger 2SAT formula such that the formula is satisfiable iff the conditions of corollaries 25, 26, 27, and 28 are met. □

RConds can be solved by reduction to 2SAT, but consider the inputs of RConds. One of them is a pattern replicator $\langle \epsilon, \alpha \rangle$ which means that for any pattern P, it must be determined if $\epsilon(P)$ and/or $\alpha(P)$. This is hardly practical. Consider, though, the possibility of specifying a finite set of patterns $\mathbb{P}$ for which $\epsilon(P)$ and $\alpha(P)$ is defined for any $P \in \mathbb{P}$. Clearly, $\mathbb{P}$ should include all the patterns occurring in the ruleset under consideration, that is, $\mathbb{P} \supseteq \bigcup_{r \in R} \mathcal{P}(r)$.

Suppose there exists $P_1, P_2 \in \mathbb{P}$ such that $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$. Then by definition 37, P_1 and P_2 are related by any P_3 such that $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$ because

$\Gamma(P_3) \subseteq \Gamma(P_1)$ and $\Gamma(P_3) \subseteq \Gamma(P_2)$. Therefore, P_3 should be included in $\mathbb{P}$ and $\epsilon(P_3)$ and $\alpha(P_3)$ should be defined.

This leaves open a number of questions, though. Even though it is clear what $\mathbb{P}$ should be, how is it actually derived (can it even be derived)? and how is it accounted for in the 2SAT reduction?

Starting with the problem of deriving $\mathbb{P}$, it must be defined how to test whether $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$, and if so, how to derive P_3 such that $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$. It is possible that there could already be a $P_4 \in \mathbb{P}$ such that $\Gamma(P_4) = \Gamma(P_3)$, and so we need a way to check whether $\Gamma(P_4) = \Gamma(P_3)$.

These problems are solved in the following lemma and proposition. Lemma 30 shows how to test whether $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$, and if it holds, how to determine P_3 such that $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$. Proposition 31 shows how to test whether $\Gamma(P_4) = \Gamma(P_3)$. Using these approaches, $\mathbb{P}$ can be derived as follows. Initialize $\mathbb{P} = \bigcup_{r \in R} \mathcal{P}(r)$. Then for every $P_1, P_2 \in \mathbb{P}$ such that $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$, determine a pattern P_3 such that $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$. Check all the $P_4 \in \mathbb{P}$ to make sure that $\Gamma(P_3) \neq \Gamma(P_4)$, and if so, add P_3 to $\mathbb{P}$. Do this iteratively until no more changes can be made to $\mathbb{P}$. Then $\mathbb{P}$ is the set of patterns such that, for all $P \in \mathbb{P}$, $\epsilon(P)$ and $\alpha(P)$ need to be defined.

Lemma 30. *Let $P_1 =$ `And(`$f_1\ x_1\ \ldots\ x_n$`)` and $P_2 =$ `And(`$f_2\ y_1\ \ldots\ y_m$`)` be patterns. If no most general unifier*[11] *σ exists for f_1 and f_2, then $\Gamma(P_1) \cap \Gamma(P_2) = \emptyset$. If no most general unifier σ exists for f_1 and f_2 such that for $1 \leq i \leq n$, $\sigma(x_i)$ is matchable, and for $1 \leq i \leq m$, $\sigma(y_i)$ is matchable, then $\Gamma(P_1) \cap \Gamma(P_2) = \emptyset$. Otherwise, let $P_3 =$ `And(`$\sigma(f_1)\ z_1\ \ldots\ z_k$`)` where $\{z_i\}_{i=1}^k$ is the maximum subset of $\{\sigma(x_i)\}_{i=1}^n \cup \{\sigma(y_i)\}_{i=1}^m$ such that every z_i is a restriction. Then $\Gamma(P_1) \cap \Gamma(P_2) = \Gamma(P_3)$.*

Proof. This proof assumes familiarity with unification and unifiers. For an overview, refer to [57].

First proving that $\Gamma(P_1) \cap \Gamma(P_2) \subseteq \Gamma(P_3)$. $f \in \Gamma(P_1) \cap \Gamma(P_2)$ iff $f \in \Gamma(P_1)$ and $f \in \Gamma(P_2)$, which is true iff there exists ground substitutions σ_1 and σ_2 such that: $\sigma_1(f_1) = f$; $\sigma_2(f_2) = f$; for $1 \leq i \leq n$, $\sigma_1(x_i)$ is matchable; and for $1 \leq j \leq m$,

[11] Note that determining a most general unifier is a well-studied and efficiently-solvable problem in computer science. [57]

$\sigma_2(y_j)$ is matchable. Then $\sigma_1 \cup \sigma_2$ is a unifier of f_1 and f_2 since $[\sigma_1 \cup \sigma_2](f_1) = f = [\sigma_1 \cup \sigma_2](f_2)$.[12] Since σ is a most general unifier of f_1 and f_2, then it holds that there is a substitution σ' such that $\sigma' \circ \sigma = [\sigma_1 \cup \sigma_2]$. So then there exists a substitution σ' such that $\sigma'(\sigma(f_1)) = f$. It also holds that $\sigma'(z_i)$ is matchable for $1 \leq i \leq k$. Therefore, $f \in \Gamma(P_3)$.

Now proving that $\Gamma(P_1) \cap \Gamma(P_2) \supseteq \Gamma(P_3)$. $f \in \Gamma(P_3)$ means that there exists a substitution σ' such that $\sigma'(\sigma(f_1)) = f$, which means that $\sigma' \circ \sigma$ is a substitution such that $\sigma' \circ \sigma(f_1) = f$. The same can be said for f_2 since σ is a unifier of f_1 and f_2 (that is, because $\sigma(f_1) = \sigma(f_2)$). It also holds that $\sigma' \circ \sigma(x_i)$ is matchable for $1 \leq i \leq n$ and that $\sigma' \circ \sigma(y_j)$ is matchable for $1 \leq j \leq m$. Therefore, $f \in \Gamma(P_1) \cap \Gamma(P_2)$. □

Proposition 31. *Let* $P_1 =$ `And(`f_1 x_1 ... x_n`)` *and* $P_2 =$ `And(`f_2 y_1 ... y_m`)` *be patterns. Remove any duplicate restrictions from* P_1*; do the same for* P_2*. Rename the variables in* P_1 *such that if the* i^{th} *term (directly) in* f_1 *is a variable, rename it* `?x`i.[13] *Do the same for* P_2*. Then using some total ordering* $\triangleleft$ *on restrictions, order the restrictions in* P_1 *and* P_2 *accordingly. Then,* $\Gamma(P_1) = \Gamma(P_2)$ *iff* $P_1 = P_2$*.*

Now to address the second question, which is, how does $\mathbb{P}$ translate into the 2SAT reduction? $\mathbb{P}$ is the set of patterns over which α and ϵ must be defined. α and ϵ, supposing to make up a valid pattern replicator, must satisfy all the inherent conditions of the definition of pattern replicator. Granted, the bulleted conditions of definition 61 are tautological, stating that for any pattern P is must hold that $\alpha(P) \vee \neg\alpha(P)$ and $\epsilon(P) \vee \neg\epsilon(P)$. Less trivial, though, is meeting the conditions of being a pattern *assignment*. These are outlined in lemma 32.

Lemma 32. *A pair of functions* $\langle \epsilon, \alpha \rangle$ *mapping patterns to boolean values constitutes a valid pattern replicator if the following hold for any pattern* P*:*

- $\alpha(P) \rightarrow \neg\epsilon(P)$*;*
- *for any pattern* P' *such that* $\Gamma(P') \subseteq \Gamma(P)$*,*

[12] Here I have assumed that P_1 and P_2 have no variables with the same name. If that is not the case, then it can be enforced by simply renaming variables in P_1 and P_2, which will not change the values of $\Gamma(P_1)$ or $\Gamma(P_2)$.

[13] Do this *simultaneously* for each variable.

- $\alpha(P) \to \alpha(P')$;
- $\epsilon(P) \to \epsilon(P')$.

Proof. Let $\langle \epsilon, \alpha \rangle$ correspond to $\langle \Phi_{\mathcal{N}}, \Omega_{\mathcal{N}}, \Theta_{\mathcal{N}} \rangle$ as defined by established notation.

Considering $\alpha(P) \to \neg\epsilon(P)$, there are three distinct cases. First, $\alpha(P) \wedge \neg\epsilon(P) \equiv [\Phi_{\mathcal{N}}(P) = \mathcal{N}] \wedge [\Omega_{\mathcal{N}}(P) = \mathcal{N}]$. Second, $\neg\alpha(P) \wedge \neg\epsilon(P) \equiv [\Phi_{\mathcal{N}}(P) = \emptyset] \wedge [\Omega_{\mathcal{N}}(P) = \mathcal{N}]$. Third, $\neg\alpha(P) \wedge \epsilon(P) \equiv [\Phi_{\mathcal{N}}(P) = \emptyset] \wedge [\Omega_{\mathcal{N}}(P) = \emptyset]$. In all cases, $\Phi_{\mathcal{N}}(P) \subseteq \Omega_{\mathcal{N}}(P) \subseteq \Theta_{\mathcal{N}}(P) = \mathcal{N}$.

Considering $\alpha(P) \to \alpha(P')$, there are three distinct cases. First, $\alpha(P) \wedge \alpha(P') \equiv [\Phi_{\mathcal{N}}(P) = \mathcal{N}] \wedge [\Phi_{\mathcal{N}}(P') = \mathcal{N}]$. Second, $\neg\alpha(P) \wedge \alpha(P') \equiv [\Phi_{\mathcal{N}}(P) = \emptyset] \wedge [\Phi_{\mathcal{N}}(P') = \mathcal{N}]$. Third, $\neg\alpha(P) \wedge \neg\alpha(P') \equiv [\Phi_{\mathcal{N}}(P) = \emptyset] \wedge [\Phi_{\mathcal{N}}(P') = \emptyset]$. In all cases, $\Phi_{\mathcal{N}}(P) \subseteq \Phi_{\mathcal{N}}(P')$.

Considering $\epsilon(P) \to \epsilon(P')$, there are three distinct cases. First, $\epsilon(P) \wedge \epsilon(P') \equiv [\Omega_{\mathcal{N}}(P) = \emptyset] \wedge [\Omega_{\mathcal{N}}(P') = \emptyset]$. Second, $\neg\epsilon(P) \wedge \epsilon(P') \equiv [\Omega_{\mathcal{N}}(P) = \mathcal{N}] \wedge [\Omega_{\mathcal{N}}(P') = \emptyset]$. Third, $\neg\epsilon(P) \wedge \neg\epsilon(P') \equiv [\Omega_{\mathcal{N}}(P) = \mathcal{N}] \wedge [\Omega_{\mathcal{N}}(P') = \mathcal{N}]$. In all cases, $\Omega_{\mathcal{N}}(P') \subseteq \Omega_{\mathcal{N}}(P)$.

Therefore, by definition 58, $\langle \epsilon, \alpha \rangle$ is a valid pattern assignment and a valid pattern replicator. □

Problem 2 (PConds)**.** For a finite set of patterns $\mathbb{P}$ and functions ϵ and α mapping patterns in $\mathbb{P}$ to boolean values, determine whether the conditions of lemma 32 are met.

Proposition 33. *PConds is reducible to 2SAT.*

Proof. By inspection of the conditions of lemma 32. □

Finally, the main theorem regarding reduction to 2SAT can be formulated. It essentially states that since the sufficient conditions for rules with replication schemes and for validity of replication schemes are all 2SAT formulas, then any assignment of variables satisfying the 2SAT formulas correspond to a replication scheme for which parallel inference (for a given polarized ruleset, using a RAOC) is correct.

Theorem 34. *Given*

- *a polarized ruleset R;*
- *a finite set of patterns $\mathbb{P}$ such that $\mathbb{P} \supseteq \bigcup_{r \in R} \mathcal{P}(r)$ and for any $P_1, P_2 \in \mathbb{P}$ such that $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$, there exists $P_3 \in \mathbb{P}$ such that $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$;*

letting

- *ψ be the 2SAT formula derived from R as described in lemma 29;*
- *γ be the 2SAT formula derived from $\mathbb{P}$ as described in proposition 33;*

then any assignment of variables in the 2SAT formula $\psi \wedge \gamma$ implies a pattern replicator $\langle \epsilon, \alpha \rangle$ such that any program $\Pi = \langle \mathcal{I}, \mathcal{H}_{fix}, S, R \rangle$ is cyclically $\mathcal{R}$-parallel where

- *$\mathcal{R}$ is any replication scheme conforming to $\langle \epsilon, \alpha \rangle$;*
- *$\mathcal{I}$ is any information keeper;*
- *S is a RAOC.*

Proof. By proposition 33, a solution to γ implies two functions ϵ' and α' mapping patterns in $\mathbb{P}$ to boolean values such that the conditions of lemma 32 are met. Then there exists a pattern replicator $\langle \epsilon, \alpha \rangle$ such that for all $P \in \mathbb{P}$, $\epsilon'(P) \equiv \epsilon(P)$ and $\alpha'(P) \equiv \alpha(P)$. Let $\mathcal{R}$ be any replication scheme that conforms to $\langle \epsilon, \alpha \rangle$. ψ is satisfied, which by lemma 29 means that the conditions of corollaries 25, 26, 27, and 28 for R are satisfied. Then by corollary 22, Π is cyclically $\mathcal{R}$-parallel. □

Corollary 35. *In addition to the conditions of theorem 34, if Π terminates, then Π is weakly $\mathcal{R}$-parallel.*

Proof. Follows quickly from theorem 34 and corollary 11. □

Having not only determined conditions under which parallel inference is correct (for a polarized ruleset with a RAOC), but having also devised a way to test those conditions, it would be instructive to test a common ruleset used to perform inference over RDF data crawled from the Semantic Web.

Table 4.1 contains the CoreRDFS ruleset, the RDFS rules that are most widely valued and supported. Reducing the ruleset to a 2SAT formula as described and then

using a SAT solver[14] to enumerate the solutions, I found there is only one solution, which corresponds to replicating all facts. It is trivially true that replicating all facts is a solution for correct parallel inference with any ruleset, but it defeats the purpose of parallelization, which is to improve performance or achieve something that is not feasible on a single machine.

Therefore, this solution is unwanted, and it can be avoided by adding another clause to the 2SAT formula. Letting $P =$ `And(?s[?p->?o])` be the pattern representing all frame facts (or triples, in the case of RDF), simply add the following clause to the 2SAT formula: $\neg\alpha(P) \vee \neg\alpha(P)$. Adding this clause states that a replication scheme is not allowed to replicate all the frame facts (or triples).

Adding that clause to the 2SAT formula derived from the CoreRDFS ruleset, it is found that there is no solution. The question then is, if correct parallel inference cannot be achieved for the CoreRDFS ruleset, then for what portion of the CoreRDFS ruleset (if any) *can* correct parallel inference be achieved? That is the topic of the following section.

Before continuing on, though, the RDFS ruleset given in table B.1 was similarly tested and no solutions were found. The same holds for the OWL2RL ruleset given in table B.6. These results should not be surprising given recent literature. Hendler and I [21] explicitly disallowed troublesome data that would cause parallel inference to be incorrect. Recently, Patel-Schneider gave a more in-depth analysis as to why (non-trivial, embarrassingly) parallel inference with the RDFS ruleset is incorrect [45].

4.2.2 Eliminating Rules by Reduction to 3SAT

The reduction to 2SAT from the previous section is useful for verifying whether a given polarized ruleset and replication scheme can result in correct parallel inference (when using a RAOC). However, in many cases, correct parallel inference is not achievable, and it is not readily apparent how to change the rules to achieve correct parallel inference.

Perhaps it can be determined which cases cause (non-trivial) parallel inference

[14]Specifically, I used relsat [58] version 2.2 [59].

Table 4.1: The CoreRDFS Ruleset

Rule ID	`If And(...)`	`Then Do(Assert(...))`
scm-spo	`?p1[rdfs:subPropertyOf->?p2]` `?p2[rdfs:subPropertyOf->?p3]`	`?p1[rdfs:subPropertyOf->?p3]`
scm-sco	`?c1[rdfs:subClassOf->?c2]` `?c2[rdfs:subClassOf->?c3]`	`?c1[rdfs:subClassOf->?c3]`
prp-spo1	`?p1[rdfs:subPropertyOf->?p2]` `?x[?p1->?y]`	`?x[?p2->?y]`
prp-dom	`?p[rdfs:domain->?c]` `?x[?p->?y]`	`?x[rdf:type->?c]`
prp-rng	`?p[rdfs:range->?c]` `?x[?p->?y]`	`?y[rdf:type->?c]`
cax-sco	`?c1[rdfs:subClassOf->?c2]` `?x[rdf:type->?c1]`	`?x[rdf:type->?c2]`

to be incorrect and, if appropriate[15], eliminate those cases. Finding cases that cause parallel inference to be incorrect can be done by modifying the reduction to 2SAT into a reduction to 3SAT.

The intuition behind the idea is rather straightforward. For every clause generated from a rule r in the reduction in the proof of lemma 29, simply add another literal to each clause, denoted $\chi(r)$. If $\chi(r)$ is assigned a value of one in the solution, then it means that rule r has been eliminated. An example will help illustrate this idea.

Consider rule prp-spo1 from table 4.1. From the reduction in the proof of lemma 29, the following formula would be generated, where $P_1 =$ `And(?s[?p1->?o])`, $P_2 =$ `And(s[?p2->?o])`, and $P_{sp} =$ `And(?p1[rdfs:subProperty->?p2])`.

$$[\alpha(P_{sp}) \vee \alpha(P_1)] \wedge [\epsilon(P_2) \vee \alpha(P_1)] \wedge [\epsilon(P_2) \vee \alpha(P_{sp})]$$

This formula represents the part of the 2SAT formula derived from rule prp-spo1 such that, when satisfied with the rest of the formula, correct parallel inference can be achieved with prp-spo1. Suppose, though, that the conditions cannot be met. Then we will want to consider the possibility of eliminating prp-spo1. Letting r be rule prp-spo1, we can change the formula to allow for that possibility.

$$\chi(r) \vee \big[[\alpha(P_{sp}) \vee \alpha(P_1)] \wedge [\epsilon(P_2) \vee \alpha(P_1)] \wedge [\epsilon(P_2) \vee \alpha(P_{sp})]\big]$$

[15] To some use case. No use case is assumed herein, but those who utilize this approach will likely not be willing to give up all cases preventing correct parallel inference.

Then by distribution, the following is equivalent.

$$[\chi(r) \vee \alpha(P_{sp}) \vee \alpha(P_1)] \wedge [\chi(r) \vee \epsilon(P_2) \vee \alpha(P_1)] \wedge [\chi(r) \vee \epsilon(P_2) \vee \alpha(P_{sp})]$$

Clearly, this is a 3SAT formula.

Using this reduction to 3SAT instead of the reduction to 2SAT from lemma 29, the overall reduction to 2SAT in theorem 34 becomes a reduction to 3SAT. Now a solution to such a 3SAT formula corresponds to a replication scheme *and* a set of rules to be eliminated, such that parallel inference will be correct (with a RAOC).

While this seems like a good idea in theory, in practice, the 3SAT formula will often have a large number of solutions. Enumerating all the solutions, or choosing an optimal solution, becomes practically intractable for non-trivial rulesets. This issue is addressed in the following section.

4.2.3 Methodology for Reducing the Search Space

In this section, a methodology is presented for reducing the search space for satisfactions of 3SAT formulas derived by rulesets and pattern replicators as demonstrated in the previous two sections. This methodology is imprecise by nature, making use of intuition and heuristics. Therefore, there is no guarantee that the methodology will provide some sort of optimal solution. Regardless, its practical value will be demonstrated by using it to restrict common rulesets for correct, parallel inference.

First, a notion of *expected factset* is needed. In the following, let an *expected factset* be a factset that is likely to require inference for some given scenario. At this point, I do not assume any particular scenario, but this notion of expected factset is helpful in capturing any intuition one might have about the factsets under consideration.

Definition 62. A pattern P is said to have high selectivity iff for any expected factset F, $|\Gamma(P) \cap F|/|F|$ is a (subjectively) small fraction.

In other words, a pattern is said to be highly selective iff it is expected that that pattern will match relatively few facts. In this way, a distinction can be made

between patterns for which replication of matched facts comes with small cost, and other patterns. Defining a few more characteristics of patterns will further help in describing the methodology steps.

Definition 63. A pattern P is said to be computable iff $\Gamma(P)$ contains only independent facts.

The notion of a computable pattern is that all the facts matched by the pattern can be determined dynamically without the need for explicit storage of facts. Thus, replication of facts matched by such patterns does not exactly result in any *physical* replication of facts.

Definition 64. A pattern P_1 is said to be related to a pattern P_2 iff $\Gamma(P_1) \cap \Gamma(P_2) \neq \emptyset$.

Definition 65. A pattern P_1 is said to be a special case of a pattern P_2 iff $\Gamma(P_1) \subseteq \Gamma(P_2)$.

Patterns are related if they both match some same fact. This means that making decisions about replication or arbitrary placement of facts matched by one pattern can have an impact on how the other pattern is classified. Special cases result in even stronger implications between patterns. These specific implications between patterns have already been proven in section 4.1. This terminology is just introduced here to simplify the description of the methodology steps.

Assume some (polarized) ruleset R under consideration. Step 1 is to assume that facts matched by selective patterns will be replicated. The idea is that, by definition, this will be a small fraction of the data, so go ahead and be liberal with replication of these facts.

Step 1. For every highly selective or computable pattern P occurring in some rule $r \in R$, replicate the facts in $\Gamma(P)$ by adding the clause $\alpha(P)$ to the SAT formula. Let $\mathbb{R}$ denote this set of patterns.

Step 2 is to restrict or "split" rules into an equivalent set of rules such that each new rule falls into one of two categories: a rule that infers facts matched by a

pattern in $\mathbb{R}$, or a rule that infers facts not matched by a pattern in $\mathbb{R}$. In essence, we are dividing rules into disjoint cases. The idea of "splitting" here is vague, but it will be illustrated in an example later in this section.

Step 2. For each rule $r \in R$, "split" r into multiple rules such that for each new rule r', all the patterns in $\mathcal{P}_A(r')$ are either special cases of patterns in $\mathbb{R}$ or unrelated to patterns in $\mathbb{R}$. Denote the new ruleset R'.

Step 3 is perhaps the most arbitrary step, although it can be quite useful for reducing the search space. It simply says to use intuition to try and determine some patterns for which matched facts should be placed arbitrarily among processors. This step can be skipped if desired, but at the risk that the search space will be less constrained (good for finding a more optimal solution, bad for tractability).

Step 3. Applying intuition, choose some patterns P that are not highly selective and allow the facts in $\Gamma(P)$ to be placed arbitrarily by adding the clause $\epsilon(P)$ to the SAT formula. Let $\mathbb{A}$ denote this set of patterns.

Step 4 simply inspects the rules to check which ones have condition formulas that can only match facts that are matched by facts in $\mathbb{R}$. In other words, these are rules for which instances will be fired by all processors. Clearly, such rules are safe for parallel inference, and so it is enforced that they not be eliminated.

Step 4. For each rule $r \in R'$, if every pattern in $\mathcal{P}_C(r)$ is a special case of a pattern in $\mathbb{R}$, then add $\neg\chi(r)$ to the SAT formula.

Step 5 is a complicated step, although more intuitive than it may appear. Simply, if a rule has no negation or retraction, and at most one of its condition patterns has not been selected for replication of facts, and every assertion is allowed to be arbitrarily placed, then do not eliminate the rule. This follows directly from corollaries 25, 26, 27, and 28.

Step 5. For each rule $r \in R'$: if $\mathcal{P}_A^-(r) = \emptyset$ and $\mathcal{P}_C^-(r) = \emptyset$; and if at most one pattern in $\mathcal{P}_C^+(r)$ is *not* a special case of a pattern in $\mathbb{R}$; and if all the patterns in $\mathcal{P}_A^+(r)$ are special cases of patterns in $\mathbb{A}$; then add $\neg\chi(r)$ to the SAT formula.

Steps 6 and 7 simply perform the reduction to 3SAT described in section 4.2.2. More interestingly, though, step 8 turns the 3SAT reduction into a heuristic Min-Cost 3SAT reduction. In the Min-Cost variations of SAT problems, variables in the SAT formulas are associated with weights (usually non-negative integers), and the goal is to find an assignment of the variables such that the formula is satisfied *and* the sum of the weights of the variables assigned a value of one, is minimized. More formally, letting $w(v)$ be the weight of a variable and $a(v)$ be the value assigned to v by assignment a, the goal is to determine an a such that the formula is satisfied *and* there is no other assignment a' such that the formula is satisfied and $\sum_v a'(v) \cdot w(v) < \sum_v a(v) \cdot w(v)$.

This "reduction" to Min-Cost 3SAT is not a true reduction because an optimal solution to the weighted 3SAT formula does not correspond to an optimal replication scheme and set of eliminated rules. This is because the variable weights are determined by some imprecise heuristics. Such heuristics are proposed later in this section.

Step 6. Initializing $\mathbb{P}$ to $\mathbb{R} \cup \mathbb{A} \cup \bigcup_{r \in R'} \mathcal{P}(r)$, iteratively add to $\mathbb{P}$ any pattern P_3 such that there exists $P_1, P_2 \in \mathbb{P}$ and $\Gamma(P_3) = \Gamma(P_1) \cap \Gamma(P_2)$ where no such P_3 already exists in $\mathbb{P}$. Do so iteratively until no changes can be made to $\mathbb{P}$.

Step 7. Generate 3SAT clauses using the modified reduction of theorem 34 as described in section 4.2.2, with R' as the ruleset and $\mathbb{P}$ as the set of patterns representing the (partial) domain for pattern replicators.

Step 8. Heuristically assign weights to the variables in the 3SAT formula, and then use the pattern replicator corresponding to an optimal solution of the 3SAT formula.

An example of the application of this methodology would be instructive. Consider the CoreRDFS ruleset from table 4.1. Let an expected factset be any factset consisting only of frames (corresponding to RDF triples) and the usual independent facts, and make the assumption that the portion of the factset constituting terminological data (or TBox in description logic vernacular) is small enough to be considered highly selective. This latter assumption is very common in the literature and has proven helpful in scaling inference to large datasets [19, 20, 21, 30, 46, 48, 51].

`rdf:type` triples are not included because they are generally not considered terminological and almost certainly constitute a relatively large portion of an expected factset.

Keep in mind, none of these steps are necessary for the reduction to correctly test the sufficient conditions for correct parallel inference. These steps are just to help reduce the search space. Prior to taking any steps, if the reduction to 3SAT is performed immediately without forcing any variable assignments, there are 809 possible solutions. Admittedly, the search space for the 3SAT formula derived from the CoreRDFS ruleset is not insurmountably large, but this is just an example to illustrate the methodology.

For step 1, select all the patterns in the CoreRDFS ruleset that are highly selective. Since it has been assumed that any pattern selecting only terminological data is highly selective, the choice is clear. $\mathbb{R}$ will consist of the following patterns.

```
And(?x1[rdfs:subPropertyOf->?x3])
And(?x1[rdfs:subClassOf->?x3])
And(?x1[rdfs:domain->?x3])
And(?x1[rdfs:range->?x3])
```

After step 1, there are 108 solutions.

For step 2, notice that for r being scm-spo or scm-sco, it naturally holds that the patterns in $\mathcal{P}_A(r)$ are special cases of the patterns in $\mathbb{R}$. For r being prp-dom, prp-rng, or cax-sco, it naturally holds that that the patterns in $\mathcal{P}_A(r)$ are unrelated to the patterns in $\mathbb{R}$. Therefore, these five rules need not be split (or rather, they are split into themselves). The only rule in need of modification is prp-spo1. It is split by restricting the variables in the action block so that each rule either produces facts that must be replicated or facts that need not necessarily be replicated. In the former case, there are four such rules.

```
If And( ?p1[rdfs:subProperty->rdfs:subPropertyOf]
        ?x[?p1->?y] )
Then Do(Assert( ?x[rdfs:subPropertyOf->?y] ))
```

```
If And( ?p1[rdfs:subProperty->rdfs:subClassOf]
        ?x[?p1->?y] )
Then Do(Assert( ?x[rdfs:subClassOf->?y] ))

If And( ?p1[rdfs:subProperty->rdfs:domain]
        ?x[?p1->?y] )
Then Do(Assert( ?x[rdfs:domain->?y] ))

If And( ?p1[rdfs:subProperty->rdfs:range]
        ?x[?p1->?y] )
Then Do(Assert( ?x[rdfs:range->?y] ))
```

For the sake of brevity, in the remainder of the paper, I will group together such rules using a newly introduced `IN` keyword (it could be considered a special builtin predicate for this very purpose, like `pred:list-contains`).

```
If And( ?p1[rdfs:subProperty->?p2]
        ?x[?p1->?y]
        ?p2 IN List(rdfs:subPropertyOf
                    rdfs:subClassOf
                    rdfs:domain
                    rdfs:range) )
Then Do(Assert( ?x[?p2->?y] ))
```

In the latter case, there is one such rule.

```
If And( ?p1[rdfs:subProperty->?p2]
        ?x[?p1->?y]
        Not(?p2 = rdfs:subPropertyOf)
        Not(?p2 = rdfs:subClassOf)
        Not(?p2 = rdfs:domain)
        Not(?p2 = rdfs:range) )
Then Do(Assert( ?x[?p2->?y] ))
```

For brevity, in the remainder of the paper, I will compress the restrictions in the condition using a newly introduced `NOTIN` keyword (this could also be considered a special builtin predicate for this very purpose).

```
If And( ?p1[rdfs:subProperty->?p2]
        ?x[?p1->?y]
        ?p2 NOTIN List(rdfs:subPropertyOf
                       rdfs:subClassOf
                       rdfs:domain
                       rdfs:range)
Then Do(Assert( ?x[?p2->?y] ))
```

After step 2, using the new ruleset R', there are 360 solutions. Note that this step actually increased the search space. This is caused by splitting the rules, which increases the number of rules and patterns, which increases the number of clauses and variables in the SAT formula, which can increase the number of solutions. Step 2 is really meant to try and preserve some of the semantics of the original ruleset by splitting the rules into cases based on whether or not they can infer (necessarily) replicated facts. Those that do must meet more conditions.

For step 3, it seems like a good idea that triples matching the following pattern should be placed arbitrarily.

```
And(?x1[?x2->?x3]
    ?x2 NOTIN List(rdfs:subPropertyOf
                   rdfs:subClassOf
                   rdfs:domain
                   rdfs:range) )
```

The intuition applied here (be it helpful or not) is that anything that is *not* considered terminological data should be placed arbitrarily. After this step, there are 128 solutions.

For step 4, rules scm-spo and scm-sco are kept from being eliminated because the patterns in their conditions are all special cases of replication patterns. For step 5, rules prp-dom, prp-rng, cax-sco, and the restricted version of prp-spo in which

inferred facts are not necessarily replicated, are all kept from elimination. Then, only one rule is at risk for elimination, and that is the following.

```
And(?x1[?x2->?x3]
    ?x2 IN List(rdfs:subPropertyOf
                rdfs:subClassOf
                rdfs:domain
                rdfs:range) )
```

After steps 4 and 5, there are only two possible solutions.

Steps 6 and 7 I have done programmatically, and in this way have been able to report the number of solutions as I progress through the steps. For the CoreRDFS ruleset, $\mathbb{P}$ contains the following patterns.

```
And(?x1[rdfs:subPropertyOf->?x3])
And(?x1[rdfs:subClassOf->?x3])
And(?x1[rdfs:domain->?x3])
And(?x1[rdfs:range->?x3])
And(?x1[rdf:type->?x3])
And(?x1[?x2->?x3]
    ?x2 NOTIN List(rdfs:subPropertyOf
                   rdfs:subClassOf
                   rdfs:domain
                   rdfs:range) )
And(?x1[?x2->?x3])
```

Even though the number of patterns is small, the SAT formula produced by step 7 has 90 clauses and is too large to display here with any clarity.

As already mentioned, there are only two solutions at this point. Since there are so few solutions, step 8 could be skipped as the two solutions can just be inspected and the preferred one chosen. However, for the purpose of example, consider the following possible heuristics.

Heuristic 1. For all $P \in \mathbb{P}$, let $weight(\epsilon(P)) = 0$.

This heuristic says that arbitrarily assigning facts to processors comes for free. It is a good thing (for parallelization) that data be kept from being replicated (if possible), so no cost should be associated with it.

Heuristic 2. For all $P \in \mathbb{P}$: if P is computable, let $weight(\alpha(P)) = 0$; otherwise, let $weight(\alpha(P)) = 1$.

Computable patterns match only facts that are not physically manifest but rather computationally determined on demand. Thus, "replication" of those facts comes for free. For other patterns, though, an arbitrary cost is associated with replication of its matched facts. This simple heuristic is not very precise. Some patterns represent a much larger number of facts than others, yet the cost for replication of facts for any non-computable pattern is the same. The heuristic is naive, but in practice, it has seemed more effective than initially expected. The reason appears to be that these patterns are often related through common subsets as a result of step 6.

Heuristic 3. For any $r \in R'$, let $weight(\chi(r)) = \sum_{P \in \mathbb{P}} weight(\alpha(P))$.

This heuristic says that it is always preferable to replicate data rather than eliminate rules, except when all data must be replicated. The assumption is that we want to preserve the originally intended semantics of the rules as much as possible.

Finally, after step 8, the selected, heuristically optimal solution corresponds to a pattern assignment and a set of rules to eliminate. Only one rule (or rather four rules shown here syntactically as one rule) is eliminated.

```
If And( ?p1[rdfs:subProperty->?p2]
        ?x[?p1->?y]
        ?p2 IN List(rdfs:subPropertyOf
                    rdfs:subClassOf
                    rdfs:domain
                    rdfs:range) )
Then Do(Assert( ?x[?p2->?y] ))
```

Furthermore, facts matching the following patterns should be replicated.

Table 4.2: The Par-CoreRDFS Ruleset

Rule ID	If And(...)	Then Do(Assert(...))
scm-spo	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p3]	?p1[rdfs:subPropertyOf->?p3]
scm-sco	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c3]	?c1[rdfs:subClassOf->?c3]
prp-spo1*	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y] ?p2 NOTIN List(rdfs:subPropertyOf rdfs:subClassOf rdfs:domain rdfs:range)	?x[?p2->?y]
prp-dom	?p[rdfs:domain->?c] ?x[?p->?y]	?x[rdf:type->?c]
prp-rng	?p[rdfs:range->?c] ?x[?p->?y]	?y[rdf:type->?c]
cax-sco	?c1[rdfs:subClassOf->?c2] ?x[rdf:type->?c1]	?x[rdf:type->?c2]

```
And(?x1[rdfs:subPropertyOf->?x3])
And(?x1[rdfs:subClassOf->?x3])
And(?x1[rdfs:domain->?x3])
And(?x1[rdfs:range->?x3])
```

In this case, they are the exact patterns that were selected for replication in step 1. Then, parallel inference with the rules in table 4.2 is correct when data is distributed (or replicated) as just stated.

At this point, it is important to draw a distinction between this work and related/previous work. In related/previous work, restrictions have been placed on the *data* (factsets) and the conclusion has been, if one does not have such data in his/her dataset (or such facts in his/her factset), then parallel inference is correct (sound and complete) [19, 21]. More concisely, correctness of inference was conditioned on characteristics of the *data*. The conditions or restrictions have been either characterized as broad propositions [21] or mired in mathematical formulas [19].

In this work, I have taken a different perspective. Instead of saying inference is correct for a ruleset R when a factset F meets certain conditions, I am determining that parallel inference is correct for a *specific* approximation of the ruleset R (which sometimes is even a very poor approximation, although not in the case of CoreRDFS and Par-CoreRDFS) *regardless* of features of the factset. More concisely, correctness of inference is conditioned on characteristics of the *rules*, and the conditions are explicitly given by determining which rules should be eliminated.

In some way, this is similar to the perspective proposed by Hitzler and van Harmelen [60], that sound and complete reasoning should be considered a gold standard, and what is really needed is a determination of how close to the standard one can come while preserving some degree of scalability. The work of this chapter represents a step in that direction by proving that under *specific*, *explicit* conditions, scalability in the form of embarrassingly parallel inference is achievable. The only question that remains is, how close is this inference to the gold standard? This remains to be determined in future work.

4.3 Deriving Rulesets Amenable to Parallel Inference

In this section, the methodology of the previous section is used to restrict the RDFS and OWL2RL rulesets into versions for which parallel inference will be correct for some non-trivial replication schemes.

4.3.1 Restricting RDFS

The RDFS ruleset is given in table B.1 in appendix B.[16] Note that this is not the complete RDFS ruleset as the infinite number of axiomatic triples related to container membership properties have been excluded in order to provide decidable inference.[17] Prior to taking any of the steps in the proposed methodology, the 3SAT formula has 1,030,268,192 solutions.

For step 1, I chose the following "terminological patterns" and computable patterns for replication.

```
And(?x1[rdfs:domain->?x3])
And(?x1[rdfs:range->?x3])
And(?x1[rdfs:subClassOf->?x3])
And(?x1[rdfs:subPropertyOf->x3])
And(?x1[rdf:type->rdfs:Class])
And(?x1[rdf:type->rdfs:Datatype])
And(?x1[rdf:type->rdfs:ContainerMembershipProperty])
```

[16] All tables prefixed with B appear in appendix B due to length of the tables.
[17] See [53] for a more thorough discussion.

```
And(External(pred:is-literal-XMLLiteral(?x1)))
And(External(pred:is-literal-PlainLiteral(?x1)))
```

Note that I have excluded the pattern `And(?x1[rdf:type->rdf:Property])` even though it could be considered terminological. This reflects some intuition on the matter. Were facts matched by `And(?x1[rdf:type->rdf:Property])` to be required to replicate, then by corollary 28, all triples would have to replicated given rule rdf1, or the rule rdf1 would have to be eliminated. Additionally, even though the individual axiomatic triples are very selective, I have not forced their replication here because they will be replicated regardless by virtue of the fact that every processor has all the rules. These finer insights reflect the imprecision of the methodology and the value of an understanding of the 3SAT reduction.

After step 1, there are 40,450,304 possible solutions, far fewer than the initial 1,030,268,192, but still a large number. Remember, though, that there is no assurance that I have not precluded an optimal solution by taking these steps, but the methodology has the benefit of making checking the sufficient conditions (via reduction to 3SAT) more tractable and efficient.

Following step 2, there are too many examples to be listed here, but for clarity, rule rdfs7 (same as prp-spo1) is a good single example of how the rules are split (again, using `IN` and `NOTIN` to syntactically compress multiple rules into fewer rules).

```
If And( ?p1[rdfs:subPropertyOf->?p2]
        ?x[?p1->?y]
        ?p2 NOTIN List(rdfs:subPropertyOf
                       rdfs:subClassOf
                       rdfs:domain
                       rdfs:range
                       rdf:type) )
Then Do(Assert( ?x[?p2->?y] ))

If And( ?p1[rdfs:subPropertyOf->?p2]
        ?x[?p1->?y]
        ?p2 IN List(rdfs:subPropertyOf
```

```
                    rdfs:subClassOf
                    rdfs:domain
                    rdfs:range) )
Then Do(Assert( ?x[?p2->?y] ))

If And( ?p1[rdfs:subPropertyOf->rdf:type]
        ?x[?p1->?y]
        ?y NOTIN List(rdfs:Class
                      rdfs:Datatype
                      rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?x[rdf:type->?y] ))

If And( ?p1[rdfs:subPropertyOf->rdf:type]
        ?x[?p1->?y]
        ?y IN List(rdfs:Class
                   rdfs:Datatype
                   rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?x[rdf:type->?y] ))
```

After step 2, there are 1,529,405,440 possible solutions, even more than at the beginning. Again, the number of solutions has grown due to the increased number of patterns and rules. Although it is not unusual for this step to increase the number of solutions, it is an important step to take so that whole rules are not eliminated simply because of some special case. In other words, more than reducing the search space, this step improves preservation of the semantics of the original rules.

For step 3, I chose to require arbitrary placement of the facts matching the following patterns.

```
And(?x1[?x2->?x3]
    ?x2 NOTIN List(rdfs:subPropertyOf
                   rdfs:subClassOf
                   rdfs:domain
                   rdfs:range
```

```
                    rdf:type) )

And(?x1[rdf:type->?x3]
    ?x3 NOTIN List(rdfs:Class
                    rdfs:Dataype
                    rdfs:ContainerMembershipProperty) )
```

After this step, there are 655,360 solutions.

Performing steps 4 and 5 programatically, there remain only 10 possible solutions. From step 6, $\mathbb{P}$ contains 87 patterns. Using the aforementioned heuristics for step 8, the final solution is as follows. Facts matched by the patterns chosen in step 1 should be replicated, and others can be placed arbitrarily. However, the following rules must be eliminated.

```
If And( ?u[rdf:type->rdf:Property] )
Then Do(Assert( ?u[rdfs:subPropertyOf->?u] ))

If And( ?x[?p->?y]
        ?p[rdfs:range->?c]
        ?c IN List(rdfs:Class
                    rdfs:Datatype
                    rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?y[rdf:type->?c] ))

If And( ?x[?p->?y]
        ?p[rdfs:domain->?c]
        ?c IN List(rdfs:Class
                    rdfs:Datatype
                    rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?x[rdf:type->?c] ))

If And( ?x[?p1->?y]
        ?p1[rdfs:subPropertyOf->?p2]
```

```
        ?p2 IN List(rdfs:subPropertyOf
                    rdfs:subClassOf
                    rdfs:domain
                    rdfs:range) )
Then Do(Assert( ?x[?p2->?y] ))

If And( ?x[?p1->?y]
        ?p1[rdfs:subPropertyOf->rdf:type]
        ?y IN List(rdfs:Class
                   rdfs:Datatype
                   rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?x[rdf:type->?y] ))

If And( ?x[rdf:type->?c1]
        ?c1[rdfs:subClassOf->?c2]
        ?c2 IN List(rdfs:Class
                    rdfs:Datatype
                    rdfs:ContainerMembershipProperty) )
Then Do(Assert( ?x[rdf:type->?c2] ))
```

Then parallel inference is correct for the Par-RDFS ruleset given in table B.5. Note that rule rdfs6 was eliminated entirely. In [21], Hendler and I reasoned that rule rdfs6 was suitable for parallel inference because the kinds of triples it infers do not – *in the context of the finite RDFS closure and with certain axiomatic assumptions on the data* – lead to further *novel* inferences that would need to be replicated. Those same axiomatic assumptions have not been assumed on the data in this work, and so that argument does not necessarily (and almost certainly does not) hold here.

4.3.2 Restricting OWL2RL

Following the methodology for the OWL2RL ruleset given in table B.6 is much more complicated given the large number of rules and patterns involved, and so in this section, just the highlights are reviewed. Note that the OWL2RL rules presented

herein are different than those from [3]. They are a RIF variation of the OWL2RL rules from [61], deviating to make the rules amenable to forward-chaining, following advice from [61] as well.

First notice that in the OWL2RL ruleset, many of the rules will get eliminated due to corollary 25. One such example is rule prp-fp.

```
If And( ?p[rdf:type->owl:FunctionalProperty]
        ?x[?p->?y1]
        ?x[?p->?y2] )
Then Do(Assert( ?y1[owl:sameAs->?y2] ))
```

The reason for direct elimination of this rule is that it contains two patterns which match all triples (frames). By corollary 25, the triples (facts) matching one of these patterns must be replicated, which means that all triples (frame facts) must be replicated. This defeats the purpose of parallelization, and so no choice remains except to eliminate rule prp-fp.

Before any steps are taken, relsat (the SAT solver that I used [59]) reports that there are 381,157,376 possible solutions, which seems too few. This report is accompanied by a warning: "WARNING: Not using a bignum package. Solution counts may overflow." Upon proceeding with step 1, forcing replication of the facts matched by the patterns in table B.8, relsat reports that there are zero possible solutions, although it enumerates many solutions. This seems to suggest that the number of solutions is so large that whatever integer datatype is being used, it is not large enough to represent the value. After step 2, relsat again reports zero possible solutions, although it provides an example solution. After forcing arbitrary placement of facts matched by patterns in table B.8, relsat again reports zero possible solutions, although it provides an example solution. After steps 4 and 5, relsat reports 180 possible solutions. From step 6, $\mathbb{P}$ contains 534 patterns.

The remaining steps pick a single solution. In short, using the rules in table B.10, replicating facts matching patterns in B.9, parallel inference is correct (when using a RAOC). Unlike with restricting the CoreRDFS and RDFS rulesets, a significant degree of the semantics represented by the original rules has been sacrificed,

which can be observed by comparing the rules in table B.6 with the rules in table B.10.

4.4 Summary

In summary, this chapter adapted the general, sufficient conditions derived in chapter 3 to apply specifically to a special case of distribution schemes called replication schemes. Determining whether these sufficient conditions are met was shown to be reducible to 2SAT. The reduction to 2SAT was then augmented to allow for the possibility to eliminate rules from the ruleset in order to improve parallelization, and the new reduction is to 3SAT. The 3SAT reduction, though, can easily result in 3SAT formulas for which there are many possible solutions, and thus, can quickly become intractable. A methodology was then proposed for how to reduce the search space for 3SAT assignments. Although the methodology does not necessarily lead to an optimal solution, it is nonetheless effective in restricting rulesets for correct parallel inference, as was illustrated by restriction of the CoreRDFS ruleset, the RDFS ruleset, and the OWL2RL ruleset.

CHAPTER 5
EVALUATION OF PARALLEL INFERENCE ON SUPERCOMPUTERS

To empirically test the theoretical findings of chapters 3 and 4, an evaluation is performed for parallel inference with rulesets that have been restricted to be amenable to parallel inference, in combination with large, established, RDF datasets.

Section 5.1 details the parameters of the evaluation, including the rulesets, datasets, metrics, and supercomputers used, as well as a discussion of the details of the (sequential) inference engine used by each processor. The bulk of the actual evaluation is reported in section 5.2.

5.1 Parameters of Evaluation

A thorough description of the parameters of the evaluation is necessary to interpret the results reported in section 5.2. This section provides such details. All code was written in C++ (except for source code from the LZO library, which is written in C) using the Message Passing Interface [62] (MPI) for interprocess communication.

5.1.1 Software Implementation of Inference Engine

In the past [21], I had used librdf [63] (aka Redland) to implement inference on an x86-based cluster, using iterative SPARQL CONSTRUCT queries as rules [64]. However, after many attempts, it did not seem feasible to install and use librdf on a Blue Gene[18] due to dependencies on other libraries. This is a common problem in using specialized supercomputers, that it is difficult to install and use such libraries on atypical architectures. For this reason, and desiring to demonstrate scaling up to large numbers of processors (like the thousands provided by a Blue Gene), I was compelled to implement an inference engine on my own. However, since a

[18]Specifically, I had tried this on a Blue Gene/L in 2009 to no avail. I am uncertain as to whether success could be had on later, more POSIX standard versions of the Blue Gene. By the time a Blue Gene/Q was made available to me, I had already written the code used in this evaluation.

sequential inference engine is not the main purpose of the work presented herein, the implementation is naive. This should be kept in mind when interpreting the relative speedup results in section 5.2 since it is well-known that relative speedup (or relative performance metrics in general) favors less efficient algorithms [65]. However, the execution times are reported as well in order to qualify any bias that may be present in the speedup results.

The inference engine uses a forward-chaining, materialization strategy. That is, starting with the data (or facts), apply the rules, and add/remove data (or inferences in the case of adding data) into the body of facts as inference progresses. The choice to perform forward-chaining here is based primarily on simplicity. Forward-chaining inference engines are easier to implement than, say, backward-chaining reasoners. The main performance drawback, though, is that forward-chaining materialization is space-intensive. Because all inferences must be stored explicitly, it is quite possible to exhaust memory, and in fact, I have easily done so with some rulesets and datasets. Regardless, the evaluation herein focuses primarily on parallel scalability, i.e., how some performance characteristic changes while varying the number of processors. Thus, efficiency of the sequential inference algorithm is secondary, and important only inasmuch as each processor must be able to perform inference. However, this decision does imply some restrictions on this evaluation, which are discussed further in this section and in section 5.1.4.

The inference engine simply iterates over the rules in the order that they are given (see section B.3), terminating when no more changes are made to the dataset (fixpoint semantics). The conflict set is never explicitly formed, and doing so is unnecessary under the semantics of a AOC because all rule instances will be fired anyway. Therefore, the rules are fired in a Datalog-like fashion, translating the rule conditions into relational queries and then using the results to fire actions. None of the rulesets under consideration (to be discussed in section 5.1.4) in this evaluation contain negation (except of safely-used equality formulas) or retraction, so order of rule firing is inconsequential, and a finite closure can always be produced.[19] The

[19]Note also that the rulesets, discussed a bit later, contain neither functions nor built-ins, and equality is used "safely." See [54, 66] for more details on the safeness of rules, which guarantees a finite closure for rulesets without negation or retraction.

queries of the condition formulas are executed from left to right in the order shown in the rule condition (i.e., a left-deep query execution plan). If at any point a join or selection results in zero results, the query returns immediately with no results. Care must be taken to make sure that equality and built-ins come after other atomic formulas that bind their variables.

Each atomic formula transforms into a select-project-rename operation, the exception being equality formulas. Equality formulas containing ground terms (this includes all restrictions as defined in definition 55) are grouped together by shared variables and executed as a single selection operation. The persistent and growing dataset (set of triples, factset) is implemented as a `std::set` with predicate-object-subject (POS) ordering. Should a triple pattern (frame formula) arise for which a POS index is unsuitable, a scan is performed on the `std::set`. Atoms are also stored in `std::set`s, each predicate having its own `std::set`, ordered lexicographically on the arguments of the atoms. As with the set of triples, when the ordering is not suited to the atom, a scan is performed. Intermediate relations are implemented as `std::deque`s, the choice of which was determined based on experience and comparison to performance using `std::vector`s and `std::list`s. Triples are implemented as fixed arrays of three terms, and tuples in intermediate results and of atoms are implemented as fixed arrays of some maximum length determined based on the specific ruleset under consideration. (The length is the maximum number of variables used in a rule, or the largest predicate arity, whichever is larger.) The data is initially dictionary-encoded (discussed in detail in appendix A), so each term is represented as a uniquely-assigned 64-bit integer.

Joins are performed using a one-sided index join. That is, of the two relations to be joined, the smaller one is indexed on the values of the join variables, and iterating over the larger relation, the index is probed for compatible tuples. The index was implemented using `std::map`, and so letting R and S be the larger and smaller relations (respectively) to be joined, building the index takes $O(|S| \cdot \lg |S|)$ time and probing it takes $O(|R| \cdot \lg |S|)$ time.

It is important to understand that although the operational semantics are prescribed in the form of forward-chained inference, that does not mean that forward-

chaining is the necessary mechanism by which inference must be performed. The findings of the previous chapters apply just as well to other mechanisms of inference inasmuch as those mechanisms effectively produce the same outcome as forward-chaining[20]. An example of this would be Datalog inference. Since Datalog programs are known to have a finite closure, then both forward-chaining and backward-chaining are capable of effectively providing the same results. From the perspective of the operational semantics in section 3.1.2, whichever mechanism is used is irrelevant, as long as from the perspective of the user, the answers come out the same as though forward-chaining was performed exactly as described in the operational semantics.[21]

5.1.2 Supercomputers and their Configurations

Two supercomputers are used in this evaluation. The first is a SMP system called Mastiff with four 16-core Opteron 6272 processors[22], 512 GB of RAM, and 2 TB of high-speed RAIDed disk. MPICH2 version 1.4.1 is the MPI version used for evaluation on Mastiff. The operating system on Mastiff was Ubuntu 12.10. The code was compiled with the `-O3` optimization flag. Additionally, when running on Mastiff, CPU affinity was set so that each processor was allowed to run on a dedicated CPU to prevent overhead of context switching and cache misses that can arise when the operating system reassigns processes to different CPUs. For example, for two processors, processor zero is assigned to CPU zero, and processor one is assigned to CPU 32. As larger numbers of processors are used, caches are increasingly shared. Cache sharing begins after four processors.

The other system is a newly installed Blue Gene/Q [22] at the CCNI. Each node of the Blue Gene/Q has 16 compute cores at 1.6 GHz each and 16 GB of RAM. Each node can be overcommitted beyond 16 tasks up to 64 tasks, although due to memory constraints, I have not taken advantage of overcommitting. The

[20] Also, if the distribution scheme is not a replication scheme, then each processor must be sure to "filter out" inferences according to θ in the distribution scheme as previously discussed in regards to line 9 of algorithm 3. Such does not apply to this evaluation since replication schemes are used.

[21] Of course, the user may perceive differences in performance, but again, *not* differences in *outcome*.

[22] Used here in the hardware sense.

nodes are interconnected by a 5D torus network. IBM's MPI compiler is used for compilation with the `-O3 -qstrict` optimization flags. When running jobs with `srun`, `--runjob-opts='--envs MALLOC_MMAP_MAX_=0 BG_MAPCOMMONHEAP=1'` was used. `MALLOC_MMAP_MAX_=0` ensures that memory does not need to be "zeroed out" when deallocated; `BG_MAPCOMMONHEAP=1` ensures that heap allocation is uniform across processors.

For both supercomputers, `posix_memalign` is used for memory allocation in place of `malloc` or similar calls (although no special allocator was written for use with STL containers).

5.1.3 Metrics

This evaluation focuses on scalability of two forms. The first is strong scaling, that is, the ability to reduce the execution time for a fixed workload by adding more processors. This is the traditional perspective of scalability from the parallel computing community, and its primary metric is speedup $S(p) = \frac{T_1}{T_p}$ where T_1 is the execution time for a single processor and T_p is the execution time for p processors. In the ideal case, $S(p) = p$, although in practice it is sometimes possible to achieve superlinear speedup when a large enough number of processors is used such that each processor's local data fits into a lower-level cache than with a smaller number of processors. There are various definitions of speedup depending on the interpretation of T_1. In absolute speedup, T_1 is the execution time for a single processor using the best sequential algorithm. However, in practice, relative speedup is often used instead, where T_1 is determined using the parallel algorithm on a single processor. Relative speedup is the strong scaling metric reported herein, although there are known issues with relative speedup favoring slower algorithms. To remedy this, execution times are reported as well. Efficiency for any form of speedup is defined as $E(p) = \frac{S(p)}{p}$, normalizing the achieved speedup to a value between zero and one[23].

An often overlooked metric that will be used in this evaluation is the Karp-Flatt metric [25]. The Karp-Flatt metric is the empirically determined serial fraction of computation. The Karp-Flatt metric provides a good indication of how parallel a

[23]Again, with exception for superlinear speedup, in which case efficiency can be greater than one.

program is, which is not always directly apparent from speedup or efficiency. This handy metric will prove quite useful in interpreting the evaluation results.

The second form of scaling is data scaling, a perspective I have proposed in [7]. Data scaling is a form of weak scaling. In weak scaling, the workload is not fixed like it is in strong scaling. Rather, it varies in some way as a function of the number of processors. Gustafson scaling [67] – also known as fixed-time scaling – scales *sequential workload* with processor size. In this case, execution time T_1 is fixed, the amount of work that p processors can do in T_1 time is determined, and then T_p is the time it takes a *single* processor to do the same amount of work that p processors did in T_1 time. So-called *scaled speedup* is then defined as $\mathcal{S}(p) = \frac{T_p}{T_1}$, and ideally $\mathcal{S}(p) = p$. Another form of weak scaling is memory-bounded scaling [68] in which the workload grows to consume the full memory capacity of the available processors. This is very close to the idea of data scaling, but it differs in that it focuses on memory capacity instead of data quantity, the two of which do not always coincide, particularly in the case of inference when one does not know *a priori* how large the closure will be.

In contrast, data scaling is concerned with the ability to handle larger quantities of data by adding more processors. In the case of inference over RDF data, the unit of data would be the RDF triple. Fixing the ratio of data quantity per processor in some sensible way to C referred to as the processor capacity, T_p is then defined as the time it takes for p processors to execute with input of size $C \cdot p$. The metric for data scaling is referred to as *growth efficiency* $\mathcal{G}(p) = \frac{T_1}{T_p}$, and ideally, $\mathcal{G}(p) = 1$, and in the worst case, $\mathcal{G}(p)$ is near zero. This metric is referred to as a form of efficiency rather than speedup because its values range between zero and one instead of zero and p, making it more similar to efficiency than speedup.

In short, the metrics for this evaluation are execution time, relative speedup, efficiency, the Karp-Flatt metric, and growth efficiency.

5.1.4 Rulesets

Although the rulesets in tables B.5 and B.10 are restricted versions of the RDFS and OWL2RL rulesets (respectively) for which parallel inference is correct,

they include some inference rules which, in practice, are troublesome and, in some cases, uninteresting. Such rules have been excluded from this evaluation. The general reason is memory constraint. Some rules lead to a drastic increase in the number of inferences, but many of these inferences appear to add only nominal value and are likely more efficiently (space-wise) inferred in a backward-chained fashion. (This has also been noted by Urbani et al. [20].)

For the restricted version of RDFS inference, the Par-CoreRDFS ruleset is used as shown in table 4.2. At first, exclusion of rules rdf1, rdfs4a, and rdfs4b was necessary (particularly the latter two) because they resulted in a large number of `?x1[rdf:type->rdfs:Resource]` and `?x1[rdf:type->rdf:Property]` triples, sometimes resulting in exhaustion of memory. Rules rdf2 and rdfs1 were also excluded due to lack of support of the `pred:is-literal-XMLLiteral` and `pred:is-literal-PlainLiteral` built-ins (defined in [69]). After eliminating those rules, keeping rules rdfs8, rdfs10, rdfs12, rdfs13, and the axiomatic triples seemed like an arbitrary compromise between the RDFS and CoreRDFS rulesets, although it would be more complete wrt the RDFS semantics [17]. Regardless, it seems more useful to consider the slightly restricted version of the CoreRDFS ruleset rather than the seemingly arbitrarily restricted version of the RDFS ruleset because the CoreRDFS ruleset represents the minimum RDFS inference support of most (if not all) RDFS inference engines.

Turning to OWL2RL, the Par-OWL2 ruleset in table B.10 represents a restricted version of the OWL2RL ruleset that is amenable to parallel inference. When initially attempting inference with the Par-OWL2 ruleset on the 2012 Billion Triples Challenge (BTC2012) dataset, it was quite easy to exhaust memory. The primary cause was rules eq-ref, eq-ref1, and eq-ref2, which essentially require the explicit statement that everything is the same as itself. For the evaluation, these rules have been excluded. For similar reason, rules scm-cls2 and scm-cls3 have been excluded. The following rules were excluded, being deemed as "uninteresting," in order to further reduce memory consumption (although the savings in some cases are admittedly quite small): scm-cls, scm-cls1, scm-op, scm-op1, scm-dp, scm-dp1, prp-ap-l, prp-ap-c, prp-ap-sa, prp-ap-idb, prp-ap-d, prp-ap-pv, prp-ap-bcw, prp-ap-iw, cls-thing,

and cls-nothing1.

The semantic purist may consider these exclusions as an arbitrary disregard for completeness, but it is not so arbitrary. This is simply an example of the delicate balance between theory and practice, and in this case, the particular difficulty in meeting the memory requirements for inference on large datasets. It is not so unusual for there to be a discrepancy between theory and practice. For example, from the point of view of theoretical computational complexity, it is often considered that a polynomial time algorithm is efficient, but in practice, if the highest power of the polynomial is sufficiently large, or if the size of the input is sufficiently large, the algorithm becomes impractical. Another example is Nick's Class (NC) for parallel complexity, which says that if a problem can be solved in polylogarithmic time with a polynomial number of processors, then it is considered to be "fast" in parallel [70]. In practice, though, developers tend not to be able to scale the number of processors polynomially. Thus, it should come as no surprise that in this case, there are practical considerations (memory) which inhibit demonstration of all theoretical findings, considering the realistic resources at one's disposal. Therefore, the discrepancy between the rulesets derived in chapter 4 and the ones used in this evaluation does not necessitate a flaw or lack of value in the findings of the previous chapters, but it does mean that the value of the rulesets derived in chapter 4 for parallel inference is *not* demonstrated by this evaluation. Overcoming the memory constraints to empirically demonstrate the value for the rulesets derived in chapter 4 remains as future work, and this evaluation instead focuses on even more restricted versions of the rulesets.

The Par-OWL2 rules excluded from the evaluation are marked with a † in table B.10. Note that the same replication scheme used for the Par-OWL2 ruleset is valid for the further restricted version since the same 3SAT reduction can be used as for deriving the Par-OWL2 ruleset, but this time, choosing the same assignment of variables with the exception that, e.g., χ(eq-ref) is set to one to indicate the elimination of rule eq-ref. Doing so still results in an assignment of variables that satisfies the 3SAT formula, which is clear by inspection of the example in section 4.2.2.

To summarize, there are two rulesets used in this evaluation: (1) the Par-CoreRDFS rules in table 4.2, and (2) the rules *not* marked with † in table B.10. The latter will be referred to as the Par-MemOWL2 ruleset.

5.1.5 Datasets

This evaluation uses two datasets that represent different ends of the spectrum in terms of difficulty. The first dataset, referred to herein as LUBM10K, is the dataset generated using the Lehigh University Benchmark (LUBM) [23] dataset generator to generate synthetic data for 10,000 universities. When generating the dataset, a random seed of zero was used (which is the common practice).

LUBM is one of the earliest, synthetic, RDF dataset generators used for performance evaluation of OWL reasoning systems. For this reason, it useful for comparing with previous systems. However, LUBM data is unrealistic. Relative to real-world datasets, there is very little data skew, and the data is quite "clean" (e.g., no chance of "ontology hijacking" [71]). For the purposes of this evaluation, though, it represents a necessary test for scalability. That is, if inference is not scalable on LUBM data, then it will very likely not be scalable for real-world datasets.

The other dataset used in this evaluation is the 2012 Billion Triples Challenge (BTC2012) dataset. The BTC2012 dataset is a set of RDF quads crawled from the Web for the purposes of a challenge in which submissions compete for the designation of champion as determined by a committee who decides, more or less (in my own determination), which submission is the most interesting considering the complexity of the dataset. The important distinction of the Billion Triples Challenge as opposed to the Open Track Challenge is that the crawled dataset must be used, and that is indeed the true challenge. The BTC datasets are well-known for having data that are troublesome (in terms of semantics [48] and performance [44]) for inference, have high data skew [44, 47, 72] and even syntactic problems [73]. For the purposes of this evaluation, BTC2012 represents a sufficient test for scalability. That is, if inference is scalable on BTC2012, then it will very likely be scalable for other real-world datasets.

These datasets have been preprocessed prior to inference, which is common in

evaluations like this (e.g., see [51]). There are three preprocessing steps for strong scaling, and there is an additional preprocessing step for data scaling. The first is normalization, applied only to BTC2012. The second is LZO compression, applied to both datasets. The third is dictionary encoding, applied to both datasets. The fourth is replication of triples as required for correct parallel inference. In the strong scaling evaluations, this step is part of the overall inference process and this is not considered preprocessing. However, in order to facilitate data scaling (the input size increases linearly with the number of processors), replication of these triples is performed prior to inference as a preprocessing step.

All steps can, in theory, be performed in parallel, and the code has been written to support it. However, given limited availability of supercomputers, normalization and LZO compression were performed on a single machine, specifically a Mac mini server with 10 GB of RAM and 768 GB of disk, 256 GB of which was a solid state drive. With such limited resources and such large datasets, normalization and LZO compression took several days for a single dataset. Dictionary encoding and replication, on the other hand, were performed on supercomputers. The rest of this section is dedicated to describing preprocessing.

In the case of BTC2012, the quad position was first stripped out to render RDF triples, and then the dataset was "normalized"[24] as in [73][25]. Normalization includes the following:

- removing lines (which should correspond to RDF triples) that are clearly invalid in N-Triples [75], which includes[26]:
  - multiple triples on one line,
  - lines with incorrectly delimited RDF terms (e.g., an IRI starting with < but not ending with >);

[24]In previous work [21, 74], I had often used a naive and simplistic N-triples parser which worked well under the assumption that the data was correctly formatted as N-triples. However, this became less useful in later BTC datasets.

[25]Note that although there were initially some bugs as reported in [73], they have been worked out prior to this evaluation.

[26]Note that non-alphanumeric blank node labels were tolerated.

- removal of lines containing invalid UTF-8 sequences[27];
- removal of lines with invalid Unicode codepoints according to UCS 6.2.0 [77];
- removal of lines with syntactically invalid IRIs according to RFC 3987 [55];
- removal of lines with invalid language tags according to RFC 5646 [78];
- normalization of IRIs according to RFC 3987 [55], which includes:
  - percent-*un*encoding any unnecessarily percent-encoded characters,
  - NFC normalization [79],
  - path resolution (e.g., `/a/../b` becomes `/b`),
  - lower-casing the scheme and host;
- NFC normalization of the lexical representations of RDF literals;
- normalization of language tags according to RFC 5646 [78].

The BTC2012 dataset was then deduplicated (that is, duplicate triples were removed). No such normalization was performed on LUBM10K.

It is important to note the distribution of the data in the N-Triples files at this point. The LUBM10K dataset is ordered as generated (triples are naturally grouped by subject) and actually contains a very small fraction of duplicates. The LUBM ontology is at the beginning of the file. The BTC2012 dataset is sorted as a result of the sequential deduplication process. These details will be important for a discussion of load-balancing in section 5.2.1.1.

After normalization, all datasets were LZO-compressed so that the data could fit within the newly instituted disk quotas at the Computational Center for Nanotechnology Innovations (CCNI) [80]. LZO compression is a block compression algorithm, and the source code from the LZO 2.06 library [81] was used. The LZO-compression approach from [72] was used to *directly* compress the N-triples data (i.e., there was no syntactic compression in the form of Sterno). A four MB block

[27] Although the BTC2012 dataset is provided as (compressed) N-Quads [76] files, which requires US-ASCII encoding, UTF-8 sequences still occur in the form of escaped sequences.

Table 5.1: Dataset Sizes

Dataset	#Triples	#Bytes				
		NT	LZO		Encoded	
			Data	Index	Data	Dict.
LUBM10K	1,334,681,485	224 GB	15.1 GB	448 KB	29.8 GB	23.0 GB
BTC2012	1,056,171,751	172 GB	17.8 GB	344 KB	23.6 GB	33.4 GB

Table 5.2: Closure Sizes in Number of (Unique) Triples

Dataset	Par-CoreRDFS	Par-MemOWL2
LUBM10K	1,667,939,414	1,950,780,866
BTC2012	2,429,721,741	(insufficient memory)

size was used with the LZO1X algorithm, and blocks were composed such that each block contained complete lines/triples. That is, no triples were split across blocks. During compression, a list of offsets into the compressed file is kept, where each offset determines the beginning of a compressed block. This allows for parallel decompression. Again, as with the normalization process, compression could in theory have been done in parallel, as demonstrated in [72], but at the time, it was more convenient to do it on a single machine, the same Mac mini server used for normalization. The LZO-compressed datasets are the versions that are deployed to the supercomputers. There, they are dictionary encoded in parallel, which is described in detail in appendix A.

Table 5.1 shows the sizes of the datasets in number of unique triples, bytes in the N-Triples file, sizes in LZO-compressed form, and sizes in dictionary compressed form. It should be noted that the LZO-compressed sizes for LUBM10K are for the LUBM10K dataset as generated (i.e., including duplicate triples, which are relatively few). The closure sizes are given in table 5.2.

Table 5.3 gives the times to dictionary encode the datasets, and table 5.4 gives the times to replicate the appropriate triples prior to data scaling. Times are reported in the format minutes:seconds. For the Blue Gene/Q, the same number of nodes was used regardless of dataset, specifically 2,048 nodes. However, for LUBM10K, 16 processors (or tasks) were assigned per node, and for BTC2012, one processor (or task) was assigned per node. The times reported are overall times, including disk I/O, with the exception of the "Repl" column in table 5.4 which

Table 5.3: Dictionary Encoding Times

System	Dataset	Ruleset	P	Min	Avg	Max	Dev
Mastiff	LUBM10K	Par-CoreRDFS	64	27:11	28:03	28:33	0:18
		Par-MemOWL2	64	28:57	29:58	30:27	0:15
	BTC2012	Par-CoreRDFS	64	38:32	43:42	46:45	1:53
Blue Gene/Q	LUBM10K	Par-CoreRDFS	32,768	0:52	1:03	1:14	0:02
		Par-MemOWL2	32,768	0:50	1:00	1:10	0:02
	BTC2012	Par-CoreRDFS	2,048	9:32	9:37	9:56	0:02

Table 5.4: Replication Preprocessing Times Prior to Data Scaling

System	Dataset	Ruleset	P	Min	Avg	Max	Dev	Repl
Mastiff	LUBM10K	Par-CoreRDFS	64	1:20	1:33	1:40	0:06	0:14
		Par-MemOWL2	64	1:55	2:05	2:11	0:06	0:50
	BTC2012	Par-CoreRDFS	64	1:15	1:24	1:29	0:03	0:38
Blue Gene/Q	LUBM10K	Par-CoreRDFS	32,768	0:11	0:20	0:24	0:02	0:07
		Par-MemOWL2	32,768	0:11	0:20	0:26	0:02	0:07
	BTC2012	Par-CoreRDFS	2,048	0:06	0:07	0:08	0:00	0:05

reports the actual time spent in performing replication (which includes querying out the triples to be replicated and the subsequent MPI calls).

Concerning dictionary encoding, it is important to note that caching of remote lookups (described in appendix A) is turned *on* for the Blue Gene/Q and turned *off* for Mastiff.

Note that no metrics are reported for Par-MemOWL2 inference on BTC2012. The reason is that preliminary evaluation using the Par-MemOWL2 ruleset with the BTC2012 dataset caused memory to become exhausted, even with 64 GB per processor (eight processors on Mastiff). Therefore, it appeared infeasible to perform such an evaluation. This is further discussed in section 5.2.

5.2 Scalability of Parallel Inference

Parallel inference consists of six phases.

1. The processors collectively read the ruleset (which has already been encoded beforehand). This step usually takes subsecond time and so is not explicitly included in the reported metrics.

2. **Load**. The processors each read the dictionary-encoded triples from their separate files and load the encoded triples into a `std::set`.

3. **Repl**. The processors replicate the data according to the predetermined pattern assignment. This is done by first collectively reading the encoded patterns from a file. Then, the triples matching the patterns are queried out of the local dataset and written into a buffer. The processors broadcast the triples to each other using `MPI::Intracomm::Allgatherv` and then load the replicated triples into their `std::set`s. This step does nothing except perform a barrier in the data scaling evaluations because in that case, replication is performed as a preprocessing step.

4. **Infer**. The processors perform inference independently of each other.

5. **Uniq**. The processors deduplicate the data by hash-distributing the triples.

6. **Write**. The processors write their local triples out to individual files.

Metrics are reported for each of the phases as appropriate (except for reading the rules, as mentioned) and for the overall execution. The replication phase and the deduplication phase act (in part) as barriers. This adds some complexity to measuring execution time. For example, the time for the replication phase of a processor begins directly after the load phase finishes. Therefore, if the load phase is unbalanced, one processor could report a longer time in the replication phase simply because it is waiting on other processors and not because any actual work is necessary. Therefore, the processor reporting the most time spent in the replication phase is often (if not always) the processor reporting the least time spent in the load phase. For this reason, the sum of the maximum times for each phase is likely greater than the overall maximum time.

Therefore, when computing metrics for each phase, for communication phases (repl, uniq), the *minimum* time is used, and for other phases (load, infer, write), the *maximum* time is used. For the overall time, the *maximum* time is used since it reflects the perceived time of the user. The same holds for charts of execution times.

5.2.1 Strong Scaling

This section is the strong scaling portion of the evaluation. Recall that strong scaling is concerned with reducing the execution time for a fixed workload by adding more processors. First, the results for Mastiff are considered, and then the Blue Gene/Q is considered. Focus should be given to the inference phase because parallelization of inference has been the focus of this thesis. Metrics for other phases are included (particularly for the overall process) in order to provide indication of other barriers to scalability that are encountered in practice.

5.2.1.1 Mastiff

Table 5.5 gives the metrics for Par-CoreRDFS inference on LUBM10K using Mastiff. The number of processors is scaled by powers of two from one to 64. While the table provides specific details, trends are more readily observed in figure 5.1.

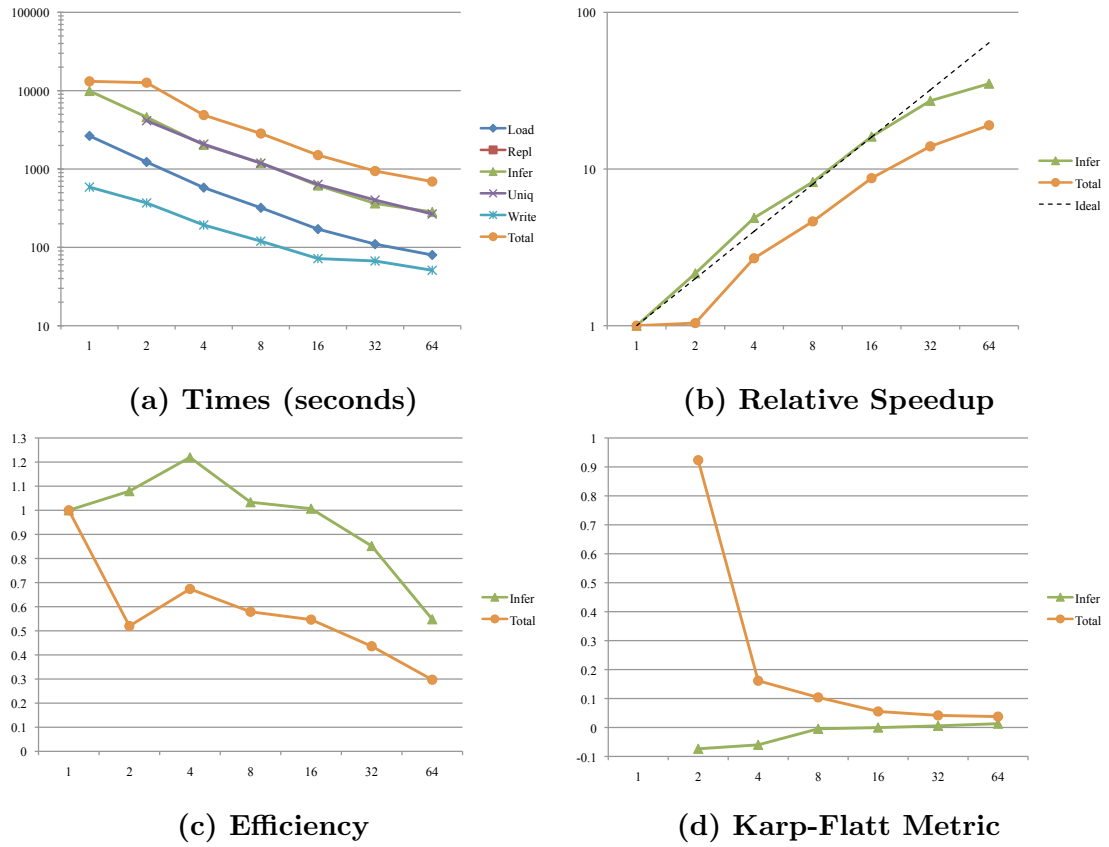
Figure 5.1a shows the execution times. Note that the time for the repl phase is not shown because it is lower than the minimum y-axis value. It is excluded to give more clarity to the phases that dominate the overall process. All phases decrease in time with number of processors. The total time does not change significantly from one to two processors because adding multiple processors introduces overhead that is non-existent for a single processor. In this case, the uniq (deduplication) phase is the specific reason for this initial lack of scalability. Also note that starting at 16 processors, disk contention when writing the output increases.

Figures 5.1b and 5.1c show the relative speedup and efficiency curves (respectively). Initially, speedup for the infer phase is superlinear up to eight processors. This is likely due to the way in which CPU affinity is set as described in section 5.1.2, reducing cache contention among processors. Thus, adding more processors with a fixed workload causes there to be less data per processor, and each processor can better utilize its cache. This is most effective at four processors when an astonishing efficiency of 1.220 is achieved for the infer phase. At eight processors, cache sharing begins, and so efficiency begins to drop.

However, after 16 processors, efficiency begins to plummet, reaching a low of 0.548 at 64 processors. The question remains as to whether this efficiency is

Table 5.5: Strong Scaling, Mastiff, Par-CoreRDFS, LUBM10K

P	Task	Min	Avg	Max	Dev	Sdup	Eff	KF
1	Load	44:08	44:08	44:08	0:00	1.000	1.000	-
	Repl	0:00	0:00	0:00	0:00	-	-	-
	Infer	2:45:22	2:45:22	2:45:22	0:00	1.000	1.000	-
	Uniq	0:00	0:00	0:00	0:00	-	-	-
	Write	9:48	9:48	9:48	0:00	1.000	1.000	-
	Total	3:39:18	3:39:18	3:39:18	0:00	1.000	1.000	-
2	Load	20:09	20:19	20:30	0:10	2.153	1.076	-0.071
	Repl	0:00	0:10	0:21	0:10	-	-	-
	Infer	1:16:36	1:16:36	1:16:36	0:00	2.159	1.079	-0.074
	Uniq	1:09:05	1:28:25	1:47:45	19:20	-	-	-
	Write	6:02	6:05	6:09	0:03	1.593	0.797	0.255
	Total	2:52:20	3:11:36	3:30:53	19:16	1.040	0.520	0.923
4	Load	9:27	9:31	9:39	0:05	4.573	1.143	-0.042
	Repl	0:00	0:08	0:12	0:05	-	-	-
	Infer	32:53	33:27	33:54	0:27	4.878	1.220	-0.060
	Uniq	34:37	35:04	35:40	0:28	-	-	-
	Write	3:03	3:07	3:13	0:04	3.047	0.762	0.104
	Total	1:21:12	1:21:18	1:21:23	0:04	2.695	0.674	0.161
8	Load	4:23	4:41	5:20	0:22	8.275	1.034	-0.005
	Repl	0:00	0:39	0:57	0:22	-	-	-
	Infer	16:28	18:15	20:00	1:39	8.268	1.034	-0.005
	Uniq	19:46	21:34	23:17	1:35	-	-	-
	Write	1:27	1:37	2:00	0:13	4.900	0.613	0.090
	Total	46:32	46:46	47:21	0:20	4.631	0.579	0.104
16	Load	2:13	2:19	2:51	0:12	15.485	0.968	0.002
	Repl	0:00	0:32	0:38	0:12	-	-	-
	Infer	7:34	8:26	10:16	0:58	16.107	1.007	0.000
	Uniq	10:36	12:27	13:18	0:56	-	-	-
	Write	0:48	0:55	1:12	0:07	8.167	0.510	0.064
	Total	24:31	24:39	25:05	0:10	8.743	0.546	0.055
32	Load	1:19	1:25	1:50	0:09	24.073	0.752	0.011
	Repl	0:00	0:25	0:31	0:09	-	-	-
	Infer	4:06	4:35	6:04	0:39	27.258	0.852	0.006
	Uniq	6:42	8:12	8:40	0:36	-	-	-
	Write	0:29	0:50	1:07	0:15	8.776	0.274	0.085
	Total	15:05	15:27	15:43	0:15	13.953	0.436	0.042
64	Load	0:52	0:56	1:20	0:09	33.100	0.517	0.015
	Repl	0:00	0:24	0:28	0:09	-	-	-
	Infer	2:20	2:48	4:43	0:47	35.060	0.548	0.013
	Uniq	4:27	6:23	6:50	0:43	-	-	-
	Write	0:24	0:41	0:51	0:07	11.529	0.180	0.072
	Total	10:54	11:13	11:32	0:09	19.014	0.297	0.038



(a) Times (seconds) **(b) Relative Speedup**

(c) Efficiency **(d) Karp-Flatt Metric**

Figure 5.1: Strong Scaling, Mastiff, Par-CoreRDFS, LUBM10K

considered good for the problem at hand. The Karp-Flatt metric can be used to answer this question.

Figure 5.1d shows the Karp-Flatt metric for Par-CoreRDFS inference on LUBM10K using Mastiff. Recall that the Karp-Flatt metric is the experimentally determined serial fraction of computation. Thus, zero is completely parallel, and one is completely serial. At two processors, the Karp-Flatt metric for the infer phase is initially negative due to superlinear speedup. The decrease in parallelization due to cache contention is apparent by the sudden jump in the metric between four and eight processors. The Karp-Flatt metric then steadily increases at a very slow rate to a value of 0.013 at 64 processors. From this, it can be concluded that although efficiency is only 0.548, it cannot be made much better than that, and the decrease in efficiency is largely inherent to the problem (that is, to Par-CoreRDFS inference on LUBM10K).

The really interesting question, though, is what makes up the 1.3% of the computation that is effectively serial? It cannot be communication because inference is embarrassingly parallel. There are three apparent (and non-mutually-exclusive) explanations. One explanation is contention for shared resources, like cache contention. Another explanation is redundant work, of which there is certainly some small amount due to rules scm-sco and scm-spo, but these rules represent a fixed amount of redundant work and cannot account for the increase in the metric. The more obvious explanation is load-balancing problems which can be easily observed from the standard deviation in the processor times reported in table 5.5. At 64 processors, the fastest processor takes 2:20 and the slowest processor takes 4:43 to perform inference, with a standard deviation of 0:47. Certainly, load-balancing is an issue.

A common and simple approach to load-balancing is static random allocation. In static random allocation, each unit of data (in this case, each triple that can be placed arbitrarily) is randomly assigned to a single processor. Preliminary experiments with static random allocation revealed some unexpected and non-intuitive findings. The goal of load-balancing is to *decrease* the maximum execution time by decreasing the standard deviation (i.e., spreading out work more evenly across processors). However, my preliminary experiments showed that static random allocation indeed decreased standard deviation, but also *increased* the maximum execution time on larger numbers of processors. Thus, the effect is the opposite of what one would expect. Additionally, memory consumption was much higher, so much so that 64 processors caused all 512 GB of Mastiff to be exhausted.

After some investigation, there is a logical explanation. Recall from section 5.1.5 that triples in the LUBM10K dataset are naturally grouped by subject. Consider the following example with subject grouping, based on characteristics of real LUBM data (using the Turtle syntax instead of RIF frames).

```
ub:teacherOf rdfs:domain ub:Faculty .
ub:Faculty rdfs:subClassOf ub:Employee .
_:prof0 ub:teacherOf _:course0 .
_:prof1 ub:teacherOf _:course1 .
```

```
_:prof1 ub:teacherOf _:course2 .
```

With the data in this order, two processors would be assigned data as follows.

```
# Processor 0
ub:teacherOf rdfs:domain ub:Faculty .
ub:Faculty rdfs:subClassOf ub:Employee .
_:prof0 ub:teacherOf _:course0 .
# Processor 1
ub:teacherOf rdfs:domain ub:Faculty .
ub:Faculty rdfs:subClassOf ub:Employee .
_:prof1 ub:teacherOf _:course1 .
_:prof1 ub:teacherOf _:course2 .
```

Then processors will draw the following inferences.

```
# Processor 0
_:prof0 rdf:type ub:Faculty .
_:prof0 rdf:type ub:Employee .
# Processor 1
_:prof1 rdf:type ub:Faculty .
_:prof1 rdf:type ub:Employee .
```

Now consider static random allocation. It is then possible that processors can be assigned data as follows.

```
# Processor 0
ub:teacherOf rdfs:domain ub:Faculty .
ub:Faculty rdfs:subClassOf ub:Employee .
_:prof1 ub:teacherOf _:course1 .
# Processor 1
ub:teacherOf rdfs:domain ub:Faculty .
ub:Faculty rdfs:subClassOf ub:Employee .
_:prof0 ub:teacherOf _:course0 .
_:prof1 ub:teacherOf _:course2 .
```

Then processors will draw the following inferences.

```
# Processor 0
_:prof1 rdf:type ub:Faculty .
_:prof1 rdf:type ub:Employee .
# Processor 1
_:prof0 rdf:type ub:Faculty .
_:prof0 rdf:type ub:Employee .
_:prof1 rdf:type ub:Faculty .
_:prof1 rdf:type ub:Employee .
```

There are two important differences to note. Firstly, even though the exact same triples are inferred, there are more replicated inferences under static random allocation than under subject grouping. This explains the increased memory consumption. Secondly, there is more redundant work. Both processors had to infer by rule scm-sco that since `_:prof1 rdf:type ub:Faculty` and `ub:Faculty rdfs:subClassOf ub:Employee`, then `_:prof1 rdf:type ub:Employee`. This explains the increased execution time. This shows that due to rules prp-dom and scm-sco, scattering triples with the same subject across processors actually results in more work. Presumably, it seems that the same could be said for prp-rng and scm-sco when triples with the same object are scattered across processors.

In short, while one would typically expect static random allocation to decrease maximum execution time by more evenly distributing data among processors, this assumption does not hold for general inference. In the case of Par-CoreRDFS inference on LUBM10K, static random allocation actually created more redundant work. It is for this reason that I have left the files in their natural ordering without any initial redistribution (except the natural side effect of the preprocessing or data generation). This suggests that load-balancing of parallel inference is non-trivial. Determining good load-balancing methods for parallel inference is left as future work. The reader should keep this caveat in mind when interpreting all results in this chapter and remember that this is the reason that explicit load-balancing has been avoided.

Urbani has previously observed this effect in his MapReduce-based inference

engine as noted in section 2.2.4 of [51]. He found that this effect made Map-side joins a less practical solution than using the Map phase for distribution and then performing joins in the Reduce phase, specifically in the context of RDFS-based inference. As a solution, Urbani distributes non-ontology triples based on terms in the triples that will be used to bind variables in rule actions. In this way, the Reduce phase will implicitly perform some (but not necessarily all) deduplication of inferences. Regardless, part of his evaluation suggests some load-balancing issues remain for real-world datasets, which is to be expected due to skew in the distribution of terms in real-world RDF data.

Returning to the discussion of the Karp-Flatt metric for Par-CoreRDFS inference on LUBM10K using Mastiff, as shown in figure 5.1d, regardless of the poor load-balancing, the infer phase is almost completely parallel. The steady (but slow) increase in the metric from eight to 64 processors can be attributed to increased contention for shared resources, poor load-balancing, and redundant work.

Now consider Par-MemOWL2 inference on LUBM10K using Mastiff. The specific metrics are given in table 5.6, but trends can be better observed in figure 5.2. The execution times are given in figure 5.2a. Note that unlike in Par-CoreRDFS inference, the repl phase takes a significant amount of time that decreases with the number of processors. This is due to an inefficiency in the way in which the data to be replicated is queried out prior to actual replication (i.e., the time is not a reflection of the cost of communication). In Par-CoreRDFS, the triples to be replicated were looked up directly via an index, but here, they are scanned out. This is an unintended side effect caused by a combination of the naive inference engine and the way in which replication patterns were specified (in rule files; see section B.3) for Par-MemOWL2. Regardless, the repl phase still takes the least amount of time, and so this inefficiency should not heavily impact the overall trend.

The speedup and efficiency curves in figures 5.2b and 5.2c, respectively, show a similar trend as with Par-CoreRDFS inference on LUBM10K. Superlinear speedup (for the infer phase) is achieved on lower numbers of processors due to little or no cache contention, and then for 32 and 64 processors, efficiency decreases significantly. At 64 processors, efficiency for the infer phase is 0.676. This brings up an inter-

Table 5.6: Strong Scaling, Mastiff, Par-MemOWL2, LUBM10K

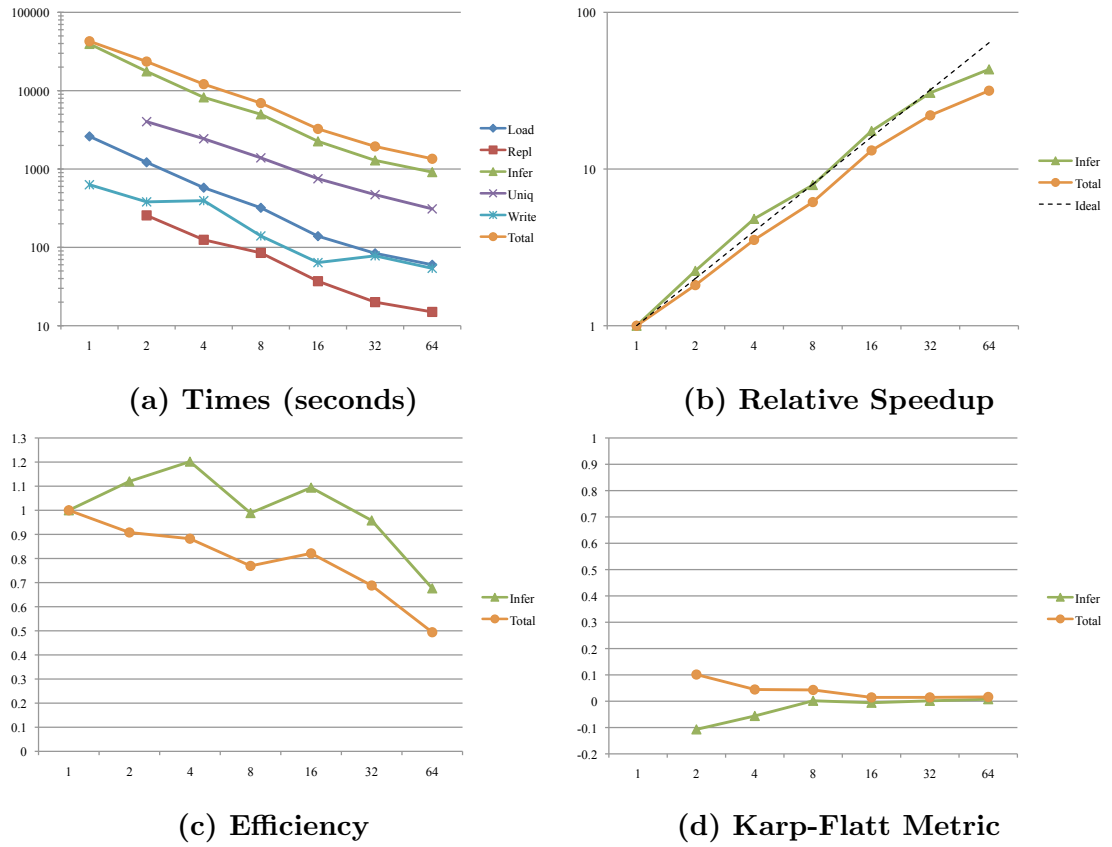
P	Task	Min	Avg	Max	Dev	Sdup	Eff	KF
1	Load	43:30	43:30	43:30	0:00	1.000	1.000	-
	Repl	0:00	0:00	0:00	0:00	-	-	-
	Infer	10:58:30	10:58:30	10:58:30	0:00	1.000	1.000	-
	Uniq	0:00	0:00	0:00	0:00	-	-	-
	Write	10:31	10:31	10:31	0:00	1.000	1.000	-
	Total	11:52:31	11:52:31	11:52:31	0:00	1.000	1.000	-
2	Load	20:13	20:15	20:18	0:02	2.143	1.071	-0.067
	Repl	4:16	4:18	4:21	0:02	-	-	-
	Infer	4:54:01	4:54:01	4:54:01	0:00	2.240	1.120	-0.107
	Uniq	1:07:08	1:07:18	1:07:28	0:10	-	-	-
	Write	6:19	6:20	6:21	0:01	1.656	0.828	0.208
	Total	6:32:04	6:32:13	6:32:22	0:09	1.816	0.908	0.101
4	Load	9:26	9:33	9:38	0:05	4.516	1.129	-0.038
	Repl	2:05	2:10	2:17	0:05	-	-	-
	Infer	2:16:42	2:16:54	2:16:58	0:07	4.808	1.202	-0.056
	Uniq	40:36	43:03	49:51	3:56	-	-	-
	Write	3:24	4:18	6:35	1:19	1.597	0.399	0.501
	Total	3:12:52	3:15:58	3:21:56	3:42	3.528	0.882	0.045
8	Load	4:26	4:35	5:20	0:17	8.156	1.020	-0.003
	Repl	1:25	2:10	2:19	0:17	-	-	-
	Infer	1:03:24	1:15:06	1:23:16	7:04	7.908	0.989	0.002
	Uniq	23:09	31:22	43:01	7:02	-	-	-
	Write	1:47	1:56	2:20	0:11	4.507	0.563	0.111
	Total	1:54:57	1:55:09	1:55:47	0:16	6.154	0.769	0.043
16	Load	2:14	2:16	2:19	0:01	18.777	1.174	-0.010
	Repl	0:37	0:40	0:42	0:01	-	-	-
	Infer	31:10	33:55	37:37	2:38	17.506	1.094	-0.006
	Uniq	12:32	16:15	18:59	2:37	-	-	-
	Write	0:58	1:01	1:04	0:02	9.859	0.616	0.042
	Total	54:03	54:07	54:13	0:03	13.142	0.821	0.014
32	Load	1:19	1:20	1:24	0:01	31.071	0.971	0.001
	Repl	0:20	0:24	0:25	0:01	-	-	-
	Infer	17:21	18:23	21:29	1:25	30.652	0.958	0.001
	Uniq	7:51	10:57	11:59	1:25	-	-	-
	Write	0:36	0:58	1:18	0:18	8.090	0.253	0.095
	Total	31:40	32:02	32:22	0:18	22.014	0.688	0.015
64	Load	0:51	0:52	1:00	0:01	43.500	0.680	0.007
	Repl	0:15	0:23	0:24	0:01	-	-	-
	Infer	10:02	11:13	15:13	1:44	43.275	0.676	0.008
	Uniq	5:10	9:10	10:21	1:44	-	-	-
	Write	0:32	0:47	0:54	0:05	11.685	0.183	0.071
	Total	22:11	22:25	22:32	0:05	31.621	0.494	0.016

esting question in comparison to the efficiency achieved with the Par-CoreRDFS ruleset, which was 0.548 at 64 processors. How is it that inference with a more complex ruleset like Par-MemOWL2 can be more efficient than with a simpler ruleset like Par-CoreRDFS? The apparent explanation is that there is proportionally more *parallel* work with Par-MemOWL2 inference. In other words, even though there is an increase in the sequential work due to an (effectively) larger ontology (i.e, more replicated TBox inference), there is a proportionally larger increase in the parallel work (i.e., even more distributed ABox inference). This is confirmed by the Karp-Flatt metric, which at 64 processors is 0.008 for Par-MemOWL2 and 0.013 for Par-CoreRDFS. This means that, non-intuitively, parallel Par-MemOWL2 inference is *more* scalable (in the strong scaling sense) than parallel Par-CoreRDFS inference, at least for LUBM data.

Again, the Karp-Flatt metric in figure 5.2d reveals that Par-MemOWL2 inference on LUBM10K is almost completely parallel, even for the total process. The increase in the metric for the inference phase from two to eight processors can be explained by poorer load-balancing. After eight processors, the inference phase appears to remain nearly completely parallel.

LUBM data is unrealistic and "easy" for inference relative to real-world datasets. Thus, the results that have been shown thus far give an indication of the scale that is achievable with the restricted rulesets (Par-CoreRDFS and Par-MemOWL2) in an ideal scenario. At this point, I turn to the real-world, BTC2012 dataset, which represents a worst-case scenario. Recall from section 5.1.5 that Par-MemOWL2 inference is not performed on BTC2012 due to memory exhaustion, although in light of the recent discussion of load-balancing, it may not be entirely impossible on Mastiff or the Blue Gene/Q if load-balancing can be improved. Another possible cause is an excessively large ontology, which cannot be remedied given the embarrassingly parallel paradigm held fixed in this evaluation. As mentioned, though, load-balancing techniques are left as future work.

The metrics for Par-CoreRDFS inference on BTC2012 (using Mastiff) are given in table 5.7. In figure 5.3a, the change in execution times are visualized. Note that the repl phase does not appear because the time is so low. A noticeable differ-



(a) Times (seconds) **(b) Relative Speedup**

(c) Efficiency **(d) Karp-Flatt Metric**

Figure 5.2: Strong Scaling, Mastiff, Par-MemOWL2, LUBM10K

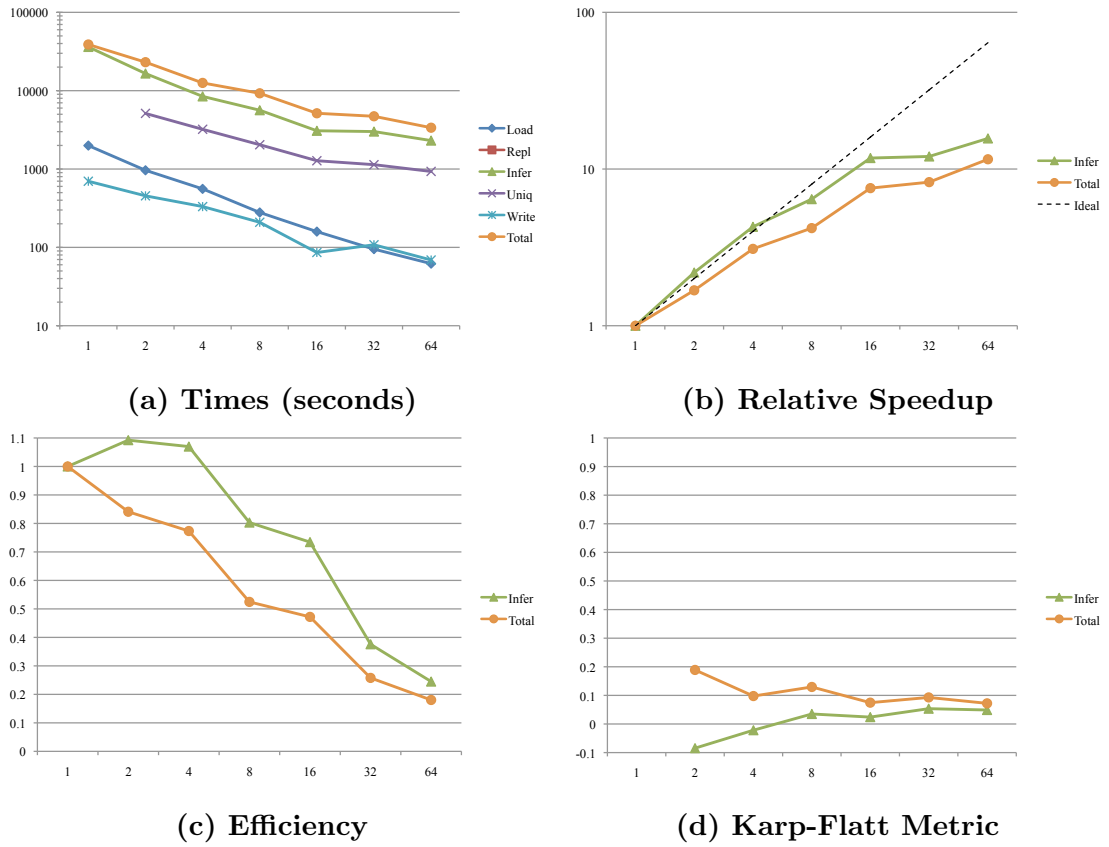
ence between Par-CoreRDFS inference on LUBM10K and the same on BTC2012 is that, with BTC2012, the infer phase dominates the total execution time. With LUBM10K, the infer phase and the uniq phase took about the same amount of time. This is because far more inferences are drawn from BTC2012 than from LUBM10K.

Speedup and efficiency, shown in figures 5.3b and 5.3c (respectively) suggest significantly poorer scalability (in the strong scaling sense) for the infer phase than with LUBM10K. Superlinear speedup is achieved at two and four processors, but that ends sharply at eight processors. Efficiency drastically decreases thereafter, ending at a low 0.245 at 64 processors. The question remains, though, whether this is good for the problem at hand.

The Karp-Flatt metric visualized in figure 5.3d indicates that, indeed, the efficiency of 0.245 is actually good at 64 processors because the infer phase is roughly 95% parallel (i.e. roughly 5% serial). As previously mentioned, the serial portion

Table 5.7: Strong Scaling, Mastiff, Par-CoreRDFS, BTC2012

P	Task	Min	Avg	Max	Dev	Sdup	Eff	KF
1	Load	33:13	33:13	33:13	0:00	1.000	1.000	-
	Repl	0:00	0:00	0:00	0:00	-	-	-
	Infer	10:03:05	10:03:05	10:03:05	0:00	1.000	1.000	-
	Uniq	0:00	0:00	0:00	0:00	-	-	-
	Write	11:38	11:38	11:38	0:00	1.000	1.000	-
	Total	10:47:56	10:47:56	10:47:56	0:00	1.000	1.000	-
2	Load	14:09	15:06	16:04	0:57	2.067	1.034	-0.033
	Repl	0:00	0:57	1:55	0:57	-	-	-
	Infer	4:36:03	4:36:03	4:36:03	0:00	2.185	1.092	-0.085
	Uniq	1:25:27	1:25:27	1:25:28	0:00	-	-	-
	Write	7:12	7:23	7:35	0:11	1.534	0.767	0.304
	Total	6:24:47	6:24:59	6:25:11	0:12	1.682	0.841	0.189
4	Load	6:16	7:19	9:18	1:11	3.572	0.893	0.040
	Repl	0:01	1:59	3:03	1:11	-	-	-
	Infer	1:44:28	2:11:49	2:20:56	15:47	4.279	1.070	-0.022
	Uniq	53:34	1:02:41	1:30:00	15:46	-	-	-
	Write	4:15	4:36	5:32	0:33	2.102	0.526	0.301
	Total	3:28:05	3:28:27	3:29:27	0:35	3.093	0.773	0.098
8	Load	2:35	3:32	4:39	0:49	7.143	0.893	0.017
	Repl	0:00	1:07	2:04	0:48	-	-	-
	Infer	48:48	1:08:22	1:33:52	16:17	6.425	0.803	0.035
	Uniq	34:03	1:04:45	1:29:47	18:51	-	-	-
	Write	1:53	2:19	3:29	0:29	3.340	0.417	0.199
	Total	2:14:10	2:20:08	2:34:23	6:43	4.197	0.525	0.129
16	Load	1:08	1:44	2:39	0:27	12.535	0.783	0.018
	Repl	0:00	0:55	1:31	0:27	-	-	-
	Infer	8:40	36:11	51:19	11:35	11.752	0.735	0.024
	Uniq	21:17	39:09	1:03:43	12:55	-	-	-
	Write	1:02	1:12	1:26	0:07	8.116	0.507	0.065
	Total	1:16:04	1:19:11	1:25:48	3:52	7.552	0.472	0.075
32	Load	0:40	0:59	1:35	0:15	20.979	0.656	0.017
	Repl	0:00	0:36	0:55	0:15	-	-	-
	Infer	6:06	23:25	50:08	10:28	12.030	0.376	0.054
	Uniq	18:57	46:43	1:02:43	10:42	-	-	-
	Write	0:37	1:08	1:48	0:28	6.463	0.202	0.127
	Total	1:11:14	1:12:51	1:18:33	2:15	8.249	0.258	0.093
64	Load	0:26	0:39	1:02	0:09	32.145	0.502	0.016
	Repl	0:00	0:23	0:36	0:09	-	-	-
	Infer	4:48	15:05	38:28	7:40	15.678	0.245	0.049
	Uniq	15:31	38:53	49:07	7:38	-	-	-
	Write	0:29	0:52	1:09	0:13	10.116	0.158	0.085
	Total	55:28	55:53	56:08	0:14	11.543	0.180	0.072



(a) Times (seconds) **(b) Relative Speedup**

(c) Efficiency **(d) Karp-Flatt Metric**

Figure 5.3: Strong Scaling, Mastiff, Par-CoreRDFS, BTC2012

of computation can be attributed to poor load-balancing (which is evident from the standard deviations in table 5.7), contention for shared resources, and redundant work.

5.2.1.2 Blue Gene/Q

The Blue Gene/Q is a very different architecture from Mastiff, as described in section 5.1.2. Most notably, it is a distributed memory architecture in which each node has 16 cores and 16 GB of memory. Strong scaling is performed for 512, 1,024, and 2,048 nodes. For inference on LUBM10K, 16 processors are assigned to a node, and so 8,192 is the base number of processors used for computing metrics. Due to the need for a large amount of memory per processor for inference on BTC2012, only one processor is assigned per node. It should be noted that, in the case of BTC2012, the computational power of the Blue Gene/Q is severely underutilized. Future work

includes adding Pthreads to the inference engine to better take advantage of the computational resources of each individual node.

For inference on LUBM10K, times are reported down to hundredths of a second so as not too lose precision in trends due to very low times. Due to a minor bug in the timing code, however, standard deviations are reported in seconds.

The metrics for strong scaling Par-CoreRDFS inference on LUBM10K using the Blue Gene/Q are given in table 5.8. Note that the time for 8,192 processors is used as the base for computing metrics. The execution times in figure 5.4a indicate that the majority of the total process is dominated by disk I/O and the uniq phase. The infer phase itself takes 4.18, 2.04, and 0.99 seconds on 8,192, 16,384, and 32,768 processors, respectively.

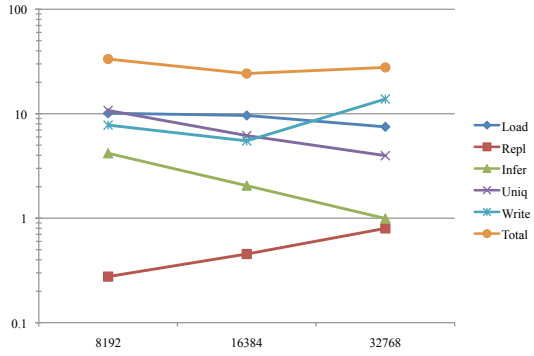
Figure 5.4b shows that, while relative speedup for the total process is terrible, the infer phase achieves superlinear speedup. This is also reflected in efficiency shown in figure 5.4c. Additionally, the Karp-Flatt metric in figure 5.4d indicates that the infer phase is completely parallel. In other words, it appears that Par-CoreRDFS inference on LUBM data is completely scalable (in the strong scaling sense) on the Blue Gene/Q. On the downside, the total process is not actually getting any faster.

Par-MemOWL2 inference on LUBM10K shows the same trends. The metrics are given in table 5.9. The execution times in figure 5.5a show that at 8,192 processors, the infer and uniq phases dominate the total process, but that changes as the number of processors increases, causing disk I/O to dominate the total process. In fact, time to write to disk increases. Superlinear speedup is still achieved for the infer phase as shown by relative speedup in figure 5.5b and efficiency in figure 5.5c. The Karp-Flatt metric in figure 5.5d shows that the infer phase is completely parallel.

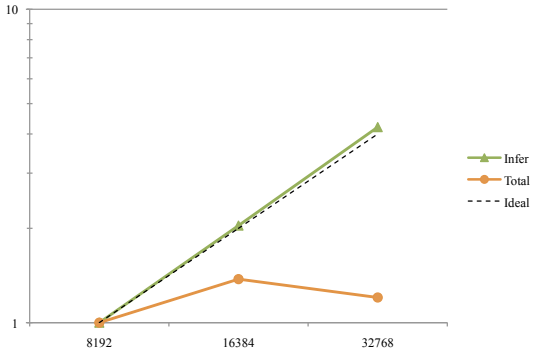
While these results on LUBM10K are exciting and quite impressive, it must be remembered that LUBM data is unrealistic and that the data is naturally organized when generated. The same cannot be said for BTC2012. The metrics for Par-CoreRDFS inference on BTC2012 are given in table 5.10. It must be noted that the maximum infer time far exceeds the average for any number of processors listed. This indicates that in the BTC2012 dataset (thinking of it as a sorted N-triples

Table 5.8: Strong Scaling, Blue Gene/Q, Par-CoreRDFS, LUBM10K

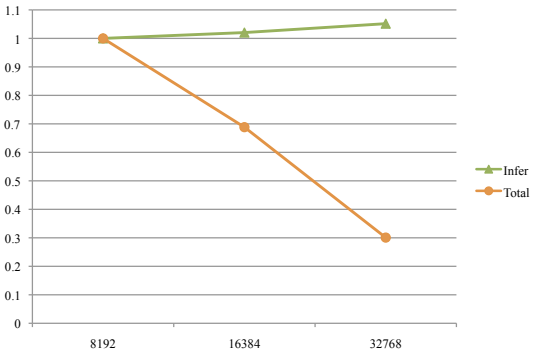
P	Task	Min	Avg	Max	Dev	$Sdup_{8192}$	Eff_{8192}	KF_{8192}
8192	Load	0:00.95	0:05.40	0:10.11	0:01	1.000	1.000	-
	Repl	0:00.27	0:04.47	0:08.47	0:01	-	-	-
	Infer	0:03.42	0:03.51	0:04.18	0:00	1.000	1.000	-
	Uniq	0:10.76	0:11.40	0:11.48	0:00	-	-	-
	Write	0:01.31	0:03.82	0:07.78	0:02	1.000	1.000	-
	Total	0:26.92	0:29.43	0:33.39	0:02	1.000	1.000	-
16384	Load	0:00.52	0:06.85	0:09.62	0:01	1.050	0.525	0.905
	Repl	0:00.45	0:02.15	0:06.43	0:01	-	-	-
	Infer	0:01.39	0:01.71	0:02.04	0:00	2.041	1.020	-0.020
	Uniq	0:06.17	0:06.50	0:06.80	0:00	-	-	-
	Write	0:01.47	0:04.80	0:05.49	0:01	1.416	0.708	0.412
	Total	0:20.24	0:23.56	0:24.25	0:01	1.377	0.688	0.453
32768	Load	0:00.26	0:05.91	0:07.49	0:01	1.350	0.337	0.655
	Repl	0:00.79	0:01.76	0:06.95	0:01	-	-	-
	Infer	0:00.43	0:00.83	0:00.99	0:00	4.207	1.052	-0.016
	Uniq	0:03.97	0:04.12	0:04.50	0:00	-	-	-
	Write	0:03.47	0:11.54	0:13.81	0:02	0.564	0.141	2.032
	Total	0:17.41	0:25.48	0:27.76	0:02	1.203	0.301	0.775



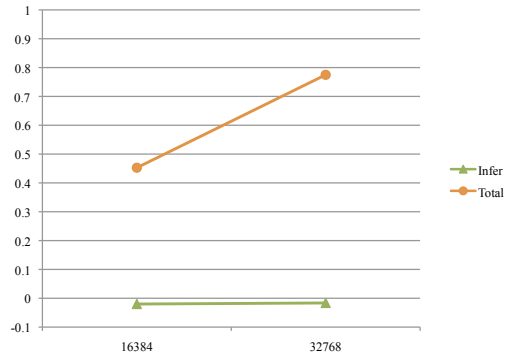
(a) Times (seconds)



(b) Relative Speedup



(c) Efficiency

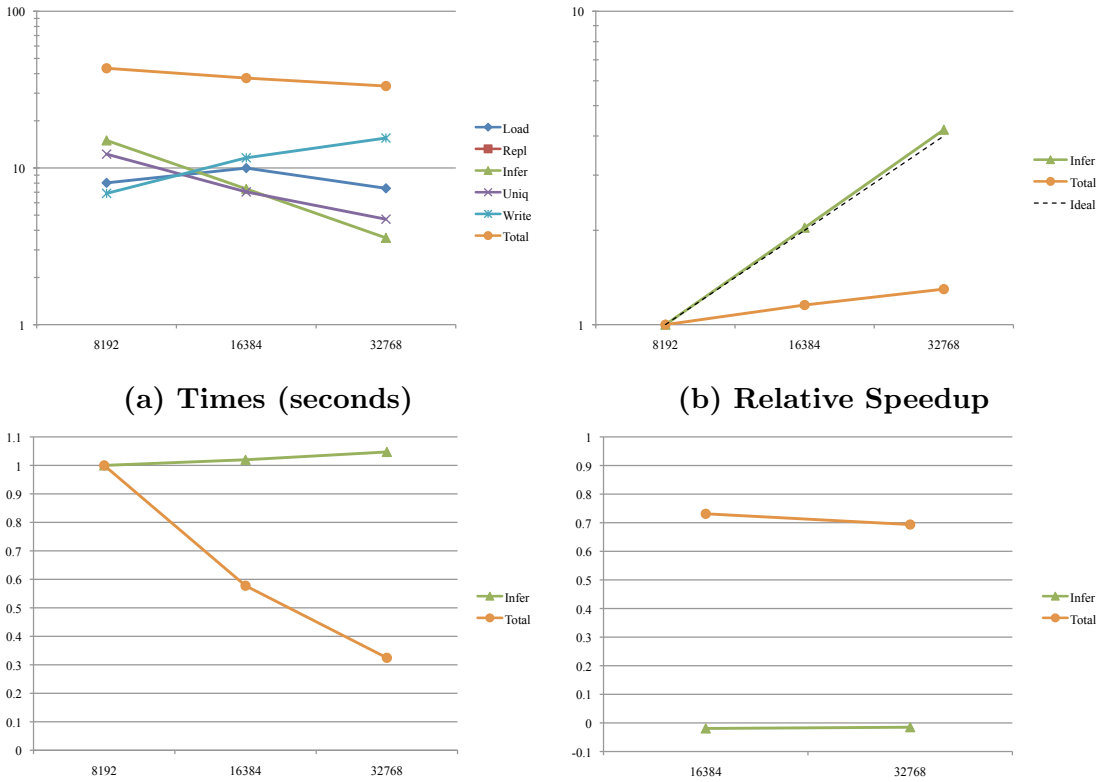


(d) Karp-Flatt Metric

Figure 5.4: Strong Scaling, Blue Gene/Q, Par-CoreRDFS, LUBM10K

Table 5.9: Strong Scaling, Blue Gene/Q, Par-MemOWL2, LUBM10K

P	Task	Min	Avg	Max	Dev	$Sdup_{8192}$	Eff_{8192}	KF_{8192}
8192	Load	0:00.83	0:04.52	0:08.03	0:06	1.000	1.000	-
	Repl	0:00.58	0:03.77	0:07.02	0:06	-	-	-
	Infer	0:12.44	0:12.70	0:14.99	0:00	1.000	1.000	-
	Uniq	0:12.25	0:14.50	0:14.76	0:00	-	-	-
	Write	0:01.04	0:03.37	0:06.88	0:02	1.000	1.000	-
	Total	0:37.44	0:39.77	0:43.28	0:02	1.000	1.000	-
16384	Load	0:00.28	0:06.87	0:09.98	0:01	0.805	0.402	1.485
	Repl	0:00.60	0:02.51	0:07.60	0:00	-	-	-
	Infer	0:05.09	0:06.19	0:07.35	0:01	2.039	1.020	-0.019
	Uniq	0:07.03	0:08.17	0:09.25	0:01	-	-	-
	Write	0:01.78	0:04.85	0:11.60	0:01	0.593	0.296	2.374
	Total	0:27.63	0:30.72	0:37.46	0:01	1.155	0.578	0.731
32768	Load	0:00.16	0:05.46	0:07.41	0:01	1.084	0.271	0.897
	Repl	0:00.91	0:01.88	0:07.60	0:01	-	-	-
	Infer	0:01.55	0:02.99	0:03.57	0:01	4.189	1.047	-0.015
	Uniq	0:04.70	0:05.28	0:06.70	0:01	-	-	-
	Write	0:02.65	0:10.71	0:15.52	0:02	0.443	0.111	2.675
	Total	0:20.49	0:28.54	0:33.33	0:02	1.298	0.325	0.694



(a) Times (seconds)

(b) Relative Speedup

(c) Efficiency

(d) Karp-Flatt Metric

Figure 5.5: Strong Scaling, Blue Gene/Q, Par-MemOWL2, LUBM10K

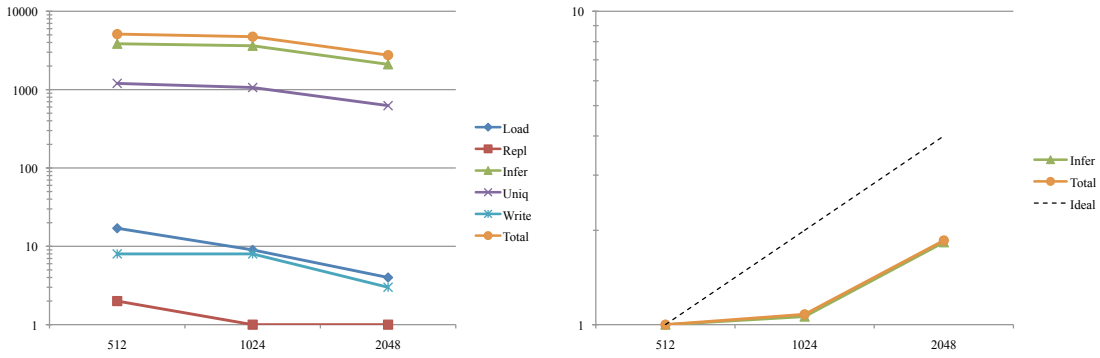
file), some regions result in far more inferences than others. This is not surprising since my colleagues and I had already witnessed such an effect in performing partial RDFS inference on the 2009 Billion Triples Challenge dataset (BTC2009) [44].

Of greatest interest, though, is that Par-CoreRDFS inference on BTC2012 with 512/1,024 processors is *slower* (considering the maximum processor time) than it is on Mastiff with 64 processors. On 2,048 processors, the maximum time is about the same. Note also that the average processor time continues to decrease. The most likely explanation is that some processor is assigned a very concentrated, "inference-rich" region of the BTC2012 dataset, and that processor is pushing its memory limits. On Mastiff, this was not an issue because processors requiring less memory allowed other processors to have more memory, and so the skew in memory requirement is tolerable due to the large shared memory. Such is not the case on the Blue Gene/Q. Better load-balancing is much needed to improve scalability on distributed memory architectures. An obvious possible solution is to allow communication between processors, taking advantage of the Blue Gene/Q's high-performance network. This is outside the scope of the study of this thesis, and so it remains as future work.

Regardless, figure 5.6 indicates poor scaling from 512 to 1,024 processors, but good scaling from 1,024 to 2,048 processors. Using 512 processors as the base for computing metrics, the infer phase is roughly 40% serial at 2,048 processors. Again, this is largely due to load-balancing issues which are not easily or intuitively solved as discussed in section 5.2.1.1. That is, the sorted order of the BTC2012 dataset has hurt here, but static random allocation would likely exacerbate the memory issues. More investigation is needed, but I conjecture that using static random allocation would give the impression of better scalability (since *relative* speedup is being used) but not necessarily faster execution time. Urbani's [51] method of data distribution may help, although it is not completely clear since load-balancing was still an issue for him on real-world datasets.

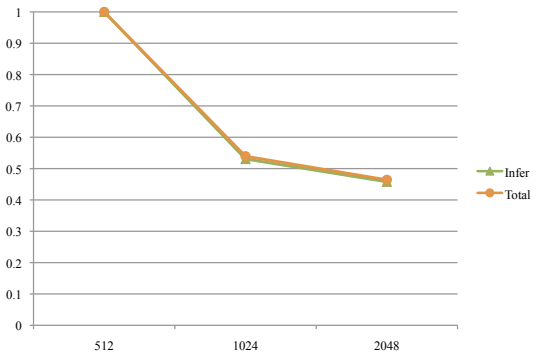
Table 5.10: Strong Scaling, Blue Gene/Q, Par-CoreRDFS, BTC2012

P	Task	Min	Avg	Max	Dev	$Sdup_{512}$	Eff_{512}	KF_{512}
512	Load	0:05	0:10	0:17	0:01	1.000	1.000	-
	Repl	0:02	0:09	0:14	0:01	-	-	-
	Infer	8:28	15:04	1:04:10	6:35	1.000	1.000	-
	Uniq	20:00	1:09:09	1:15:43	6:33	-	-	-
	Write	0:03	0:06	0:08	0:01	1.000	1.000	-
	Total	1:24:33	1:24:38	1:25:13	0:03	1.000	1.000	-
1024	Load	0:02	0:05	0:09	0:01	1.889	0.944	0.059
	Repl	0:01	0:05	0:08	0:01	-	-	-
	Infer	8:11	12:22	1:00:28	4:27	1.061	0.531	0.885
	Uniq	17:44	1:05:51	1:10:01	4:26	-	-	-
	Write	0:02	0:04	0:08	0:01	1.000	0.500	1.000
	Total	1:18:23	1:18:26	1:18:59	0:02	1.079	0.539	0.854
2048	Load	0:00	0:03	0:04	0:01	4.250	1.062	-0.020
	Repl	0:01	0:03	0:06	0:01	-	-	-
	Infer	8:02	10:53	35:02	1:44	1.832	0.458	0.395
	Uniq	10:25	34:35	37:25	1:44	-	-	-
	Write	0:01	0:01	0:03	0:00	2.667	0.667	0.167
	Total	45:34	45:35	45:54	0:01	1.857	0.464	0.385

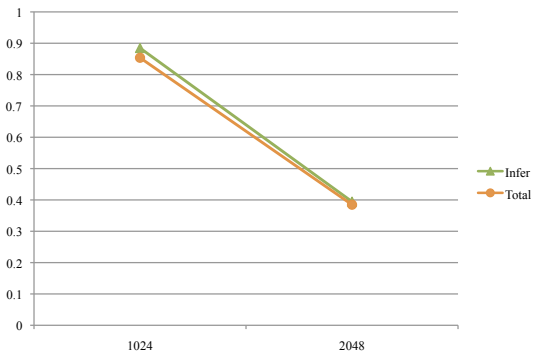


(a) Times (seconds)

(b) Relative Speedup



(c) Efficiency



(d) Karp-Flatt Metric

Figure 5.6: Strong Scaling, Blue Gene/Q, Par-CoreRDFS, BTC2012

5.2.2 Data Scaling

Now I turn to a different kind of scalability called data scaling which has already been introduced in section 5.1.3. In short, though, data scaling is concerned with how well execution time can be held constant (or rather, kept from increasing) as processors are added, *holding the ratio of input size to number of processors fixed*. This perspective is data-centric in that it is concerned with handling more data, not executing faster. Thus, it is a form of weak scaling.

The metric for data scaling is called growth efficiency [7]. A value of one indicates that scaling number of processors linearly with input size holds execution time constant. A value less than one indicates that adding more processors cannot handle (linearly) larger input without increasing the execution time.

The data scaling evaluation is performed by first having the maximum number of processors for the given scenario perform inference on the entire dataset. Then half of the processors perform inference on the first half of the dataset. Then a quarter of the processors perform inference on the first quarter of the dataset, and so on. Thus, data scaling is very susceptible to distribution skew in the dataset files, and this should be kept in mind when interpreting all results in this section.

Recall from section 5.1.5 that the replication phase is performed as a preprocessing step for data scaling. Thus, the times for repl reported in the tables and figures simply indicate how long processors waited at a barrier.

5.2.2.1 Mastiff

Data scaling on Mastiff is not entirely sensible. As an SMP machine, scaling up processors within a single machine does not provide additional (space-related) resources for handling more data. Regardless, a data scaling evaluation can reveal characteristics about the data and the machine that can provide more insights into the ruleset and dataset under consideration.

The metrics for Par-CoreRDFS inference on LUBM10K are given in table 5.11. Note that even though the task is effectively the same for 64 processors as it was in strong scaling, the times are generally faster in the data scaling results. The greatest contributing factor to this difference appears to be that loading the data was faster.

Table 5.11: Data Scaling, Mastiff, Par-CoreRDFS, LUBM10K

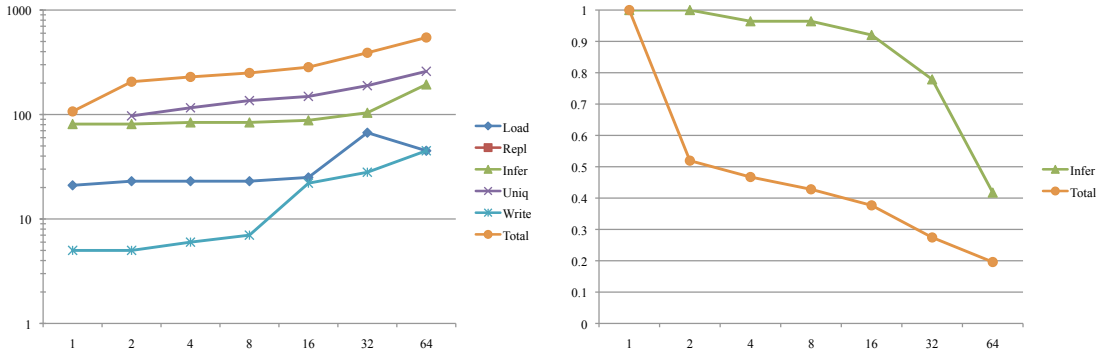
P	Task	Min	Avg	Max	Dev	GEff
1	Load	0:21	0:21	0:21	0:00	1.000
	Repl	0:00	0:00	0:00	0:00	-
	Infer	1:21	1:21	1:21	0:00	1.000
	Uniq	0:00	0:00	0:00	0:00	-
	Write	0:05	0:05	0:05	0:00	1.000
	Total	1:47	1:47	1:47	0:00	1.000
2	Load	0:22	0:22	0:23	0:00	0.913
	Repl	0:00	0:00	0:01	0:00	-
	Infer	1:21	1:21	1:21	0:00	1.000
	Uniq	1:37	1:37	1:37	0:00	-
	Write	0:05	0:05	0:05	0:00	1.000
	Total	3:26	3:26	3:26	0:00	0.519
4	Load	0:22	0:23	0:23	0:00	0.913
	Repl	0:00	0:00	0:01	0:00	-
	Infer	1:22	1:23	1:24	0:01	0.964
	Uniq	1:56	1:57	1:58	0:01	-
	Write	0:06	0:06	0:06	0:00	0.833
	Total	3:49	3:49	3:49	0:00	0.467
8	Load	0:21	0:23	0:23	0:01	0.913
	Repl	0:00	0:00	0:02	0:01	-
	Infer	1:22	1:23	1:24	0:01	0.964
	Uniq	2:16	2:17	2:18	0:01	-
	Write	0:06	0:07	0:07	0:00	0.714
	Total	4:09	4:10	4:10	0:00	0.428
16	Load	0:23	0:24	0:25	0:01	0.840
	Repl	0:00	0:01	0:02	0:01	-
	Infer	1:25	1:27	1:28	0:01	0.920
	Uniq	2:29	2:30	2:32	0:01	-
	Write	0:07	0:21	0:22	0:04	0.227
	Total	4:29	4:43	4:44	0:04	0.377
32	Load	0:26	0:46	1:07	0:19	0.313
	Repl	0:00	0:21	0:41	0:19	-
	Infer	1:38	1:42	1:44	0:02	0.779
	Uniq	3:09	3:11	3:15	0:02	-
	Write	0:09	0:25	0:28	0:03	0.179
	Total	6:11	6:27	6:30	0:03	0.274
64	Load	0:31	0:33	0:45	0:02	0.467
	Repl	0:00	0:12	0:14	0:02	-
	Infer	2:20	2:45	3:14	0:14	0.418
	Uniq	4:19	4:49	5:13	0:13	-
	Write	0:20	0:34	0:45	0:04	0.111
	Total	8:42	8:54	9:06	0:04	0.196

Table 5.12: Data Scaling, Mastiff, Par-MemOWL2, LUBM10K

P	Task	Min	Avg	Max	Dev	GEff
1	Load	0:22	0:22	0:22	0:00	1.000
	Repl	0:00	0:00	0:00	0:00	-
	Infer	5:05	5:05	5:05	0:00	1.000
	Uniq	0:00	0:00	0:00	0:00	-
	Write	0:05	0:05	0:05	0:00	1.000
	Total	5:32	5:32	5:32	0:00	1.000
2	Load	0:21	0:21	0:22	0:00	1.000
	Repl	0:00	0:00	0:01	0:00	-
	Infer	5:06	5:07	5:08	0:01	0.990
	Uniq	1:58	1:59	2:00	0:01	-
	Write	0:06	0:06	0:06	0:00	0.833
	Total	7:34	7:34	7:34	0:00	0.731
4	Load	0:22	0:22	0:23	0:00	0.957
	Repl	0:00	0:01	0:01	0:00	-
	Infer	5:07	5:07	5:08	0:00	0.990
	Uniq	2:21	2:21	2:22	0:00	-
	Write	0:07	0:07	0:07	0:00	0.714
	Total	7:59	7:59	7:59	0:00	0.693
8	Load	0:21	0:23	0:24	0:01	0.917
	Repl	0:00	0:01	0:03	0:01	-
	Infer	5:04	5:35	6:20	0:35	0.803
	Uniq	2:43	3:28	3:59	0:35	-
	Write	0:08	0:08	0:11	0:01	0.455
	Total	9:35	9:35	9:38	0:01	0.574
16	Load	0:22	0:23	0:26	0:01	0.846
	Repl	0:00	0:03	0:04	0:01	-
	Infer	5:21	5:55	7:13	0:46	0.704
	Uniq	2:57	4:15	4:49	0:45	-
	Write	0:09	0:12	0:14	0:02	0.357
	Total	10:45	10:48	10:51	0:02	0.510
32	Load	0:27	0:27	0:29	0:01	0.759
	Repl	0:00	0:01	0:02	0:01	-
	Infer	6:16	7:13	10:00	1:20	0.508
	Uniq	3:47	6:34	7:31	1:19	-
	Write	0:12	0:24	0:34	0:09	0.147
	Total	14:28	14:40	14:50	0:09	0.373
64	Load	0:31	0:32	0:39	0:02	0.564
	Repl	0:00	0:07	0:08	0:02	-
	Infer	7:49	11:02	14:55	2:43	0.341
	Uniq	5:05	8:59	12:11	2:42	-
	Write	0:21	0:39	0:47	0:08	0.106
	Total	21:00	21:19	21:26	0:07	0.258

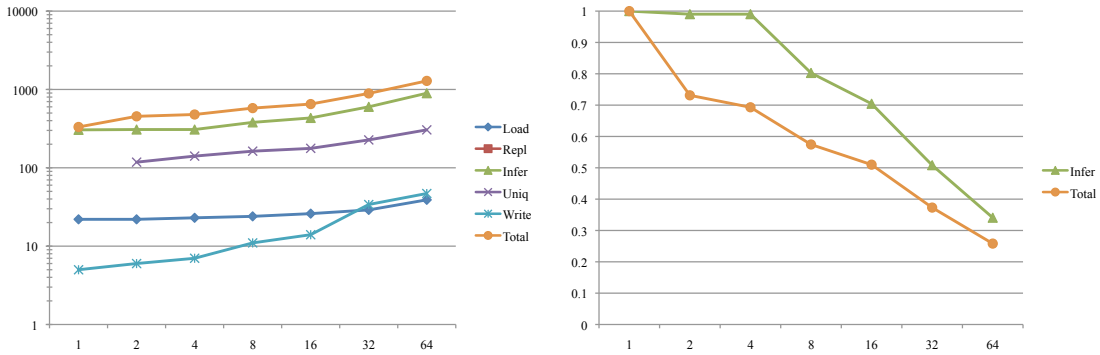
Table 5.13: Data Scaling, Mastiff, Par-CoreRDFS, BTC2012

P	Task	Min	Avg	Max	Dev	GEff
1	Load	0:17	0:17	0:17	0:00	1.000
	Repl	0:00	0:00	0:00	0:00	-
	Infer	9:47	9:47	9:47	0:00	1.000
	Uniq	0:00	0:00	0:00	0:00	-
	Write	0:09	0:09	0:09	0:00	1.000
	Total	10:13	10:13	10:13	0:00	1.000
2	Load	0:17	0:17	0:17	0:00	1.000
	Repl	0:00	0:00	0:00	0:00	-
	Infer	6:03	7:55	9:47	1:52	1.000
	Uniq	2:51	4:42	6:33	1:51	-
	Write	0:08	0:08	0:08	0:00	1.125
	Total	13:01	13:02	13:03	0:01	0.783
4	Load	0:17	0:17	0:18	0:00	0.944
	Repl	0:00	0:00	0:01	0:00	-
	Infer	6:04	7:38	9:49	1:22	0.997
	Uniq	3:29	5:39	7:13	1:21	-
	Write	0:08	0:08	0:09	0:00	1.000
	Total	13:43	13:43	13:45	0:01	0.743
8	Load	0:16	0:17	0:18	0:01	0.944
	Repl	0:00	0:01	0:02	0:01	-
	Infer	6:05	7:24	9:59	1:03	0.980
	Uniq	3:58	6:32	7:51	1:02	-
	Write	0:09	0:10	0:10	0:00	0.900
	Total	14:23	14:24	14:25	0:00	0.709
16	Load	0:15	0:19	0:24	0:03	0.708
	Repl	0:00	0:05	0:09	0:03	-
	Infer	4:42	12:41	20:11	6:28	0.485
	Uniq	7:14	14:46	22:43	6:27	-
	Write	0:12	0:14	0:16	0:01	0.562
	Total	28:02	28:05	28:08	0:02	0.363
32	Load	1:19	1:29	1:44	0:07	0.163
	Repl	0:00	0:15	0:25	0:07	-
	Infer	3:50	12:23	32:54	8:16	0.297
	Uniq	11:18	31:48	40:19	8:14	-
	Write	0:15	0:29	0:40	0:09	0.225
	Total	46:13	46:23	46:36	0:09	0.219
64	Load	0:20	0:26	0:33	0:03	0.515
	Repl	0:00	0:07	0:13	0:03	-
	Infer	4:53	15:19	39:25	7:59	0.248
	Uniq	15:29	39:33	49:56	7:57	-
	Write	0:29	0:45	0:57	0:07	0.158
	Total	55:55	56:10	56:20	0:07	0.181



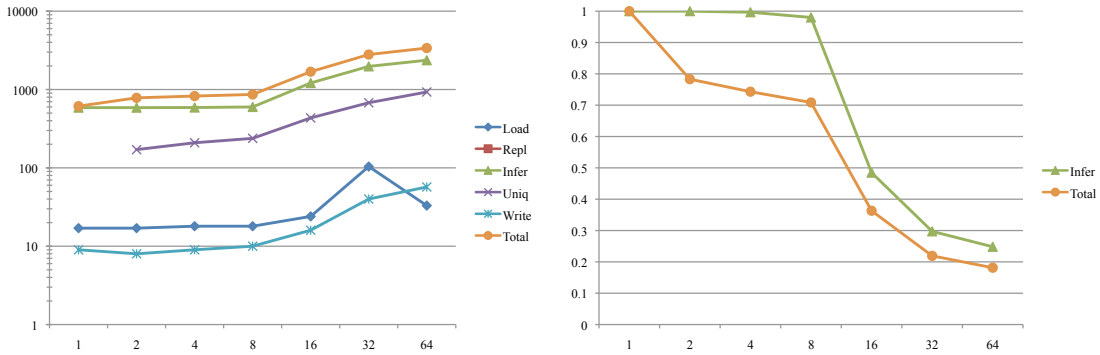
(a) Times (seconds) (b) Growth Efficiency

Figure 5.7: Data Scaling, Mastiff, Par-CoreRDFS, LUBM10K



(a) Times (seconds) (b) Growth Efficiency

Figure 5.8: Data Scaling, Mastiff, Par-MemOWL2, LUBM10K



(a) Times (seconds) (b) Growth Efficiency

Figure 5.9: Data Scaling, Mastiff, Par-CoreRDFS, BTC2012

This could be explained by the fact that each processor's local input dataset is sorted as a result of having performed the replication phase as a preprocessing step, and thus loading the triples into a `std::set` is faster. Of course, the repl phase, which is really just a barrier, is also faster.

Trends can be observed in figure 5.7. Note that the infer phase appears to keep its time fairly constant until 16-32 processors. At 64 processors, growth efficiency for the infer phase is a mere 0.418. This is an expected result, and it confirms two things. First, the LUBM10K dataset has a fairly even distribution to it wrt Par-CoreRDFS inference. If it did not, then growth efficiency would change more drastically (in either direction) on lower numbers of processors. Second, since the LUBM10K dataset is fairly evenly distributed, the contention for resources is the only explanation for the decrease in growth efficiency.

For Par-MemOWL2 inference on LUBM10K, the metrics are given in table 5.12 and visualized in figure 5.8. The infer phase takes nearly the same amount of time up to four processors, and after that, the time begins to increase. This makes sense because, at eight processors, sharing of L2 cache begins. However, the significant increase in time was not observed for Par-CoreRDFS inference until after 16 processors. This suggests that with the smaller number of inferences for Par-CoreRDFS inference, processors can take greater advantage of their (independent) L1 caches and more efficiently share their L2 caches. With Par-MemOWL2, there are more inferences, and so cache contention increases. This is clearly observed by the steady decline of growth efficiency from four to 64 processors.

For Par-CoreRDFS inference on BTC2012, the metrics are given in table 5.13 and visualized in figure 5.9. The infer phase is nearly constant up to eight processors, and then the time increases drastically after that. Although cache contention is a factor, this behavior seems abnormal compared to inference on LUBM10K. Figure 5.9b shows that from eight to 16 processors, there is a steep drop in growth efficiency, much steeper than was witnessed with LUBM10K. An explanation could be skew in the data. Data scaling on the Blue Gene/Q will help to determine the root cause since scaling number of nodes does not increase cache contention.

5.2.2.2 Blue Gene/Q

Unlike with Mastiff, data scaling is more sensible on a distributed memory architecture in which increasing the number of *independent*[28] nodes can handle larger datasets without concern for contention of shared resources.

Times for inference on LUBM10K are reported down to hundredths of a second in order to preserve precision in the trends. Due to a bug in the timing code, standard deviation of times is not reported for LUBM10K.

The metrics for data scaling Par-CoreRDFS inference on LUBM10K are given in table 5.14. As is expected, figure 5.10 indicates nearly perfect data scaling for the infer phase. For Par-MemOWL2 (metrics in table 5.15), figure 5.11 also indicates good data scaling, but not close perfect. This is difficult to explain because data scaling is perfect from 8,192 to 16,384 processors, so this would suggest that, given the even distribution of LUBM data and the fact that inference is embarrassingly parallel, it should also hold for higher numbers of processors. However, figure 5.11b contradicts this intuition with a drop in growth efficiency from 16,384 to 32,768 processors. Further investigation is needed and is left as future work.

Turning to BTC2012, the metrics for data scaling Par-CoreRDFS inference are given in table 5.16. Figure 5.12a shows that time for the infer phase remains essentially constant from 512 to 1,024 processors and then suddenly increases from 1,024 to 2,048 processors. Growth efficiency for the infer phase in figure 5.12b is poor for 2,048 processors at a mere 0.442. This is the result of skew in the distribution of data in BTC2012. Specifically, it appears the the second half of the dataset contains more work than the first half. Here the sorted order of the dataset has hurt again. Recall, though, that were I to use static random allocation, memory would be exhausted, and inference would be infeasible. Future work includes further study on load-balancing including experiments using Urbani's approach [51]. Considering the high-performance network of the Blue Gene/Q, the speed-dating approach of Kotoulas et al. is also a good candidate for consideration [47].

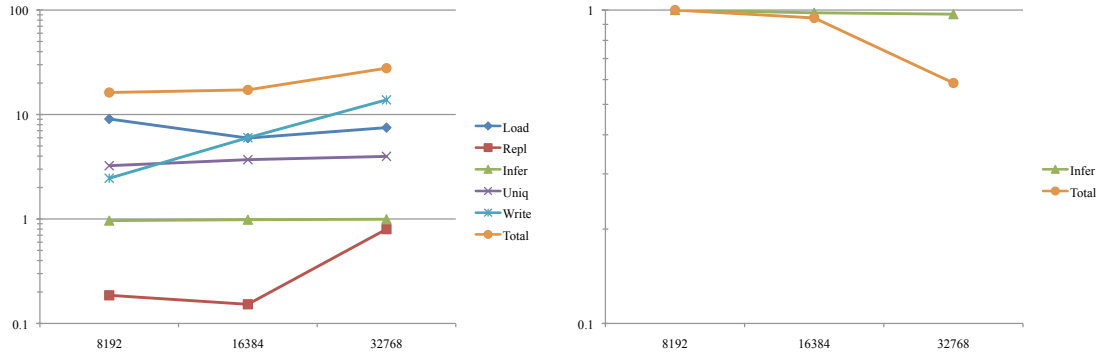
[28] Independent with respect to embarrassingly parallel computation.

Table 5.14: Data Scaling, Blue Gene/Q, Par-CoreRDFS, LUBM10K

P	Task	Min	Avg	Max	$GEff_{8192}$
8192	Load	00:00.25	00:03.51	00:09.06	1.000
	Repl	00:00.18	00:05.37	00:09.02	-
	Infer	00:00.85	00:00.88	00:00.96	1.000
	Uniq	00:03.23	00:03.28	00:03.35	-
	Write	00:00.57	00:01.83	00:02.45	1.000
	Total	00:14.36	00:15.62	00:16.24	1.000
16384	Load	00:00.24	00:03.58	00:05.94	1.523
	Repl	00:00.15	00:02.02	00:05.80	-
	Infer	00:00.85	00:00.89	00:00.98	0.980
	Uniq	00:03.69	00:03.76	00:03.83	-
	Write	00:01.52	00:04.94	00:05.99	0.409
	Total	00:12.73	00:16.16	00:17.21	0.944
32768	Load	00:00.26	00:05.91	00:07.49	1.210
	Repl	00:00.79	00:01.76	00:06.95	-
	Infer	00:00.43	00:00.83	00:00.99	0.970
	Uniq	00:03.97	00:04.12	00:04.50	-
	Write	00:03.47	00:11.54	00:13.81	0.177
	Total	00:17.41	00:25.48	00:27.76	0.585

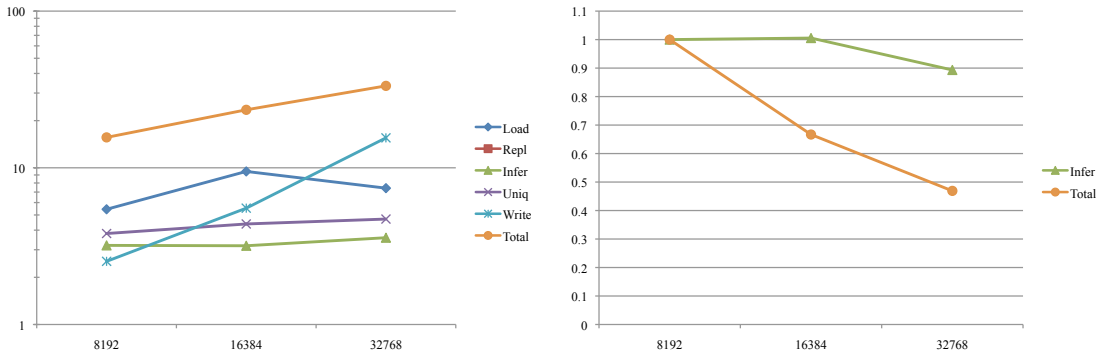
Table 5.15: Data Scaling, Blue Gene/Q, Par-MemOWL2, LUBM10K

P	Task	Min	Avg	Max	$GEff_{8192}$
8192	Load	00:00.21	00:03.46	00:05.43	1.000
	Repl	00:00.13	00:01.63	00:05.39	-
	Infer	00:02.96	00:03.05	00:03.19	1.000
	Uniq	00:03.80	00:03.92	00:04.03	-
	Write	00:00.56	00:01.87	00:02.52	1.000
	Total	00:13.67	00:14.98	00:15.63	1.000
16384	Load	00:00.25	00:03.86	00:09.48	0.573
	Repl	00:00.17	00:04.92	00:09.28	-
	Infer	00:02.96	00:03.05	00:03.18	1.005
	Uniq	00:04.38	00:04.48	00:04.59	-
	Write	00:01.54	00:04.93	00:05.52	0.458
	Total	00:19.45	00:22.85	00:23.44	0.667
32768	Load	00:00.16	00:05.46	00:07.41	0.733
	Repl	00:00.91	00:01.88	00:07.60	-
	Infer	00:01.55	00:02.99	00:03.57	0.893
	Uniq	00:04.70	00:05.28	00:06.70	-
	Write	00:02.65	00:10.71	00:15.52	0.163
	Total	00:20.49	00:28.54	00:33.33	0.469



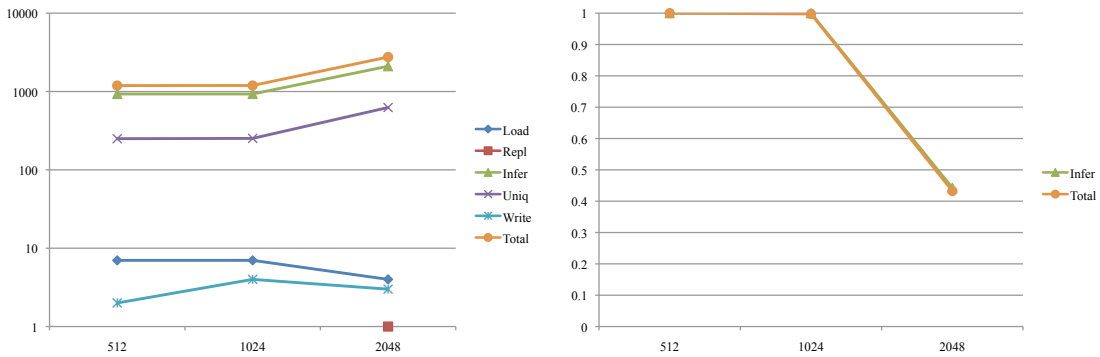
(a) Times (seconds) (b) Growth Efficiency

Figure 5.10: Data Scaling, Blue Gene/Q, Par-CoreRDFS, LUBM10K



(a) Times (seconds) (b) Growth Efficiency

Figure 5.11: Data Scaling, Blue Gene/Q, Par-MemOWL2, LUBM10K



(a) Times (seconds) (b) Growth Efficiency

Figure 5.12: Data Scaling, Blue Gene/Q, Par-CoreRDFS, BTC2012

Table 5.16: Data Scaling, Blue Gene/Q, Par-CoreRDFS, BTC2012

P	Task	Min	Avg	Max	Dev	$GEff_{512}$
512	Load	0:03	0:05	0:07	0:01	1.000
	Repl	0:00	0:02	0:04	0:01	-
	Infer	8:11	11:24	15:30	1:23	1.000
	Uniq	4:10	8:17	11:29	1:22	-
	Write	0:01	0:01	0:02	0:00	1.000
	Total	19:47	19:48	19:51	0:01	1.000
1024	Load	0:03	0:05	0:07	0:01	1.000
	Repl	0:00	0:03	0:05	0:01	-
	Infer	8:09	11:05	15:30	1:15	1.000
	Uniq	4:12	8:37	11:33	1:14	-
	Write	0:00	0:01	0:04	0:00	0.500
	Total	19:50	19:51	19:54	0:01	0.997
2048	Load	0:00	0:03	0:04	0:01	1.750
	Repl	0:01	0:03	0:06	0:01	-
	Infer	8:02	10:53	35:02	1:44	0.442
	Uniq	10:25	34:35	37:25	1:44	-
	Write	0:01	0:01	0:03	0:00	0.667
	Total	45:34	45:35	45:54	0:01	0.432

5.3 Summary

This evaluation set out to test the scalability that can be achieved with restricted rulesets such that correct (embarrassingly) parallel inference is possible. Using LUBM10K, a very high degree of scalability was achieved for Par-CoreRDFS and Par-MemOWL2 inference in both a strong scaling and data scaling scenario. The Karp-Flatt metric revealed that inference on LUBM10K is almost completely parallel in strong scaling, and growth efficiency revealed that good data scaling is also achievable.

The importance of the results with LUBM10K is that they concretely demonstrate that the theoretical results of highly parallel and scalable inference from previous chapters are achievable in practice and not merely some unachievable mathematical ideal. However, LUBM10K is ideal. It does not contain many of the practical difficulties that are encountered in real-world datasets.

Therefore, parallel inference was also performed on the BTC2012 dataset. Par-MemOWL2 inference on BTC2012 resulted in memory exhaustion, and so it appears that in some cases, a huge amount of memory is necessary for parallel inference to even be feasible. Par-CoreRDFS inference on BTC2012 was feasible and demon-

streated excellent strong scaling up to 64 processors on Mastiff, a SMP system. However, poor scalability was observed on the Blue Gene/Q. This is attributed to skew in the dataset that causes some processors to need far more memory than others. On Mastiff, this was not an issue because processors needing more memory could use memory not used by processors needing less memory. In the distributed environment of the Blue Gene/Q, such was not possible, and the processors needing a large amount of memory were likely pushing their capacities and hurting performance.

These problems could theoretically be solved by effective load-balancing. Using a simple solution like static random allocation, though, actually hurts overall performance and increases memory consumption in contrast to the natural subject grouping of the datasets as generated, as discussed in section 5.2.1.1. Load-balancing is left as future work. Candidate approaches include those used by Urbani [51] and Kotoulas et al. [47].

In short, parallel inference on Semantic Web data is memory-intensive and relies heavily on load-balancing for scalability. If these issues are solvable, then it appears from the evaluation on LUBM10K that a high degree of scalability can be achieved.

Additionally, using a Blue Gene/Q, inference for interesting Semantic Web rulesets has been demonstrated on 1.33 billion LUBM triples around 30 seconds and on the BTC2012 dataset around 45 minutes. The fastest time for Par-CoreRDFS inference on LUBM10K reported herein is 28 seconds, a few seconds faster than the current record (to the best of my knowledge) of 31 seconds achieved on a Cray XMT [82] by Goodman and Mizell [33]. To the best of my knowledge, this work is also the first to have performed any kind of well-defined, complete closure[29] on a BTC dataset, although Williams et al. [44] performed a kind of partial RDFS closure on the BTC2009 dataset.

[29]To be clear, this is the complete Par-CoreRDFS closure wrt the operational semantics from chapter 3 and *not* the model-theoretic semantics of RDFS from [17]. The former is a specific subset of the latter.

CHAPTER 6
CONCLUSION

Inference on the Semantic Web continues to be a problem due in part to the large volume of data involved. This thesis has addressed the problem from a perspective of data parallelism, attempting to scale production rule inference to larger datasets by adding more processors. However, the achievable degree of parallelism is not merely a function of clever implementation. The rules and data also directly impact the degree of parallelism that can be achieved as demonstrated in chapter 5.

Previous work focused on determining restrictions on the data such that correctness of parallel inference could be ensured, but in reality, on the Web, no such restrictions can be reasonably enforced. Thus, it makes more sense to consider conditions and restrictions on the rules. Another way to look at this is that, instead of maximizing expressivity and suffering whatever performance inhibitions come with it, consider fixing the requisite performance characteristics and instead suffer a loss of expressivity.

In chapter 3, the definitions and operational semantics for the production rules under consideration were given, and definitions for parallel inference were introduced. Following that, sufficient conditions were determined for ground rules such that, when the conditions are met for every rule instance of a rule in a ruleset, parallel inference is correct with respect to a distribution scheme. These findings are restricted to a certain class of rulesets referred to as polarized rulesets in which each rule has either only assert actions or only retract actions. They are also restricted to a certain class of CRSs called RAOCs in which all rule instances are fired in every cycle such that retractions precede assertions. These conditions were then generalized to rules (not just ground rules) and patterns (instead of individual facts) so that rules could be directly tested.

In chapter 4, a specific form of distribution schemes was considered called replication schemes. Using replication schemes is significantly simpler and has nice properties such as triviality of implementing parallel inference and ease of specifi-

cation. The sufficient conditions for rules were then reformulated to be specific to replication schemes, which led to the observation that testing these new conditions is reducible to satisfiability, and not just SAT, but specifically 2SAT. The 2SAT reduction is useful for testing conditions and for deriving replication schemes that support correct parallel inference for a (polarized) ruleset (with a RAOC), but many interesting rulesets will have only a single solution: to replicate all facts to all processors. Therefore, the 2SAT reduction was augmented into a 3SAT reduction that allows for the possibility to eliminate rules in order to improve parallelization. The downside, though, is that even for moderately sized rulesets, the search space for solutions to the 3SAT formulas can be quite large. Therefore, a methodology was also given to aide in deriving restricted rulesets amenable to parallel inference. This methodology was then used to derive restricted versions of the RDFS and OWL2RL rulesets.

In chapter 5, an evaluation was performed to demonstrate the scalability that is achievable using restricted versions of the RDFS and OWL2RL rulesets. Two large datasets were used, referred to as LUBM10K and BTC2012. The LUBM10K is an unrealistic, synthetic dataset of over 1.3 billion triples, and BTC2012 is a real-world dataset crawled from the Web containing over one billion triples. The evaluation demonstrated that, when there is sufficient memory and sufficient load-balancing is achieved, a very high degree of parallelism can be achieved. This was made obvious by use of the Karp-Flatt metric (the experimentally determined serial fraction of computation) and an SMP machine. The real difficulty, though, arises in a distributed memory environment, where load-balancing and large memory per node becomes essential. Scalability of inference was demonstrated on LUBM10K up to 32,768 processors on 2,048 nodes of a Blue Gene/Q, achieving inference times (not including other phases) of no more than a few seconds.

To pithily (and roughly) summarize the overall contribution of this thesis, I have provided proof and methodology for determining parts of production rule inference that are embarrassingly parallel, and I have demonstrated that when practical issues of load-balancing and memory availability can be solved, a high degree of availability can be achieved. Future work should focus on addressing the practical

issues that prevent achieving a high degree of scalability, and addressing portions of inference that are not embarrassingly parallel.

6.1 Future Work

Although this thesis constitutes a significant contribution toward webscale inference, it is really only a first step in the right direction. Conditions have been determined under which, as has been demonstrated, highly scalable inference is feasible. However, these conditions are quite restrictive, e.g. disallowing functional and inverse functional properties from OWL2RL. The reason is that throughout this work, one characteristic has been held fixed: embarrassingly parallel computation. That is, processors have not been allowed to communicate with each other during inference.

The natural, next step is then to introduce communication in order to handle rules that had to be eliminated in order to preserve correct, (embarrassingly) parallel inference. In this case, the 3SAT reduction can simply be reinterpreted. Specifically, instead of interpreting $\chi(r)$ to mean the elimination of rule r, let it instead mean that (some of) the inferences of r must be (dynamically) replicated.

The evaluation in chapter 5, while sufficient for the purposes of this thesis, left much to be desired. Consideration of a "middle ground" dataset would likely be helpful to those who have real-world data that is not quite so difficult as BTC2012. As mentioned several times, investigation into effective load-balancing is needed for distributed memory architectures. Additionally, for inference over BTC2012, the computational power of the Blue Gene/Q was severely underutilized. A more parallel-aware inference engine utilizing Pthreads or OpenMP would resolve this issue.

Some supercomputers are not amenable to embarrassing parallelism and are optimized for cases that necessitate interaction of processors. For example, the Cray XMT [82] has large shared memory without a typical cache hierarchy, and so dividing up the problem to be local to individual processors does not result in significant performance gains relative to more typical architectures. However, the Cray XMT (hardware) processor is designed to avoid penalties from accessing memory, and so

its strength relative to more typical architectures is in problems that are the *opposite* of embarrassing parallelism. Thus, parallelizing inference for such architectures is a very different problem and requires an entirely different perspective and approach than presented in this thesis.

Finally, as mentioned, I was not able to perform Par-MemOWL2 inference on the BTC2012 dataset due to an explosion of inferences that quickly exhausted memory. Hogan et al. surmised that this can be due (at least in part) to abuse of ontologies, termed "ontology hijacking" [19, 48]. They propose restrictions on inference in order to avoid such abuses, thus reducing the number of inferences. Such approaches are likely necessary, and more are needed. Therefore, it remains as future work to determine root causes for the explosion of inferences, and to figure out appropriate ways to cope with them.

LITERATURE CITED

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific Amer.*, vol. 284, no. 5, pp. 28–37, May 2001.

[2] G. Klyne and J. J. Carroll. (2004, Feb.). Resource description framework (RDF): Concepts and abstract syntax. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ [Retrieved Feb. 12, 2013]

[3] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. (2009, Oct.). OWL 2 web ontology language profiles. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/ [Retrieved Feb. 12, 2013]

[4] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge, United Kingdom: Cambridge University Press, 2003.

[5] M. Kifer and H. Boley. (2010, Jun.). RIF overview. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/NOTE-rif-overview-20100622/ [Retrieved Feb. 12, 2013]

[6] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989.

[7] J. Weaver, "A scalability metric for parallel computations on large, growing datasets (like the web)," in *Proc. Joint Workshop Scalable and High-Performance Semantic Web Systems*, Boston, MA, Nov. 2012, pp. 91–96.

[8] D. P. Miranker, "Special issue on the parallel execution of rules systems: Guest editor's introduction," *J. Parallel and Distributed Computing*, vol. 13, no. 4, pp. 345–347, Dec. 1991.

[9] S. Kuo and D. Moldovan, "The state of the art in parallel production systems," *J. Parallel and Distributed Computing*, vol. 15, no. 1, pp. 1–26, May 1992.

[10] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: A survey," *ACM Trans. Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, Jul. 2001.

[11] W. Zhang, K. Wang, and S.-C. Chau, "Data partition and parallel evaluation of datalog programs," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 1, pp. 163–176, Feb. 1995.

[12] O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule-processing," in *Proc. 1990 ACM SIGMOD Int. Conf. Management of Data*, Atlantic City, NJ, 1990, pp. 133–142.

[13] O. Wolfson and A. Silberschatz, "Distributed processing of logic programs," in *Proc. 1988 ACM SIGMOD Int. Conf. Management of Data*, Chicago, IL, 1988, pp. 329–336.

[14] J. G. Schmolze, "Guaranteeing serializable results in synchronous parallel production systems," *J. Parallel and Distributed Computing*, vol. 13, no. 4, pp. 348–365, Dec. 1991.

[15] T. Ishida and S. J. Stolfo, "Towards the parallel execution of rules in production system programs," in *Proc. Int. Conf. Parallel Processing*, 1985, pp. 568–575.

[16] J. N. Amaral and J. Ghosh, "A concurrent architecture for serializable production systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 12, pp. 1265–1280, Dec. 1996.

[17] P. Hayes. (2004, Feb.). RDF semantics. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ [Retrieved Feb. 12, 2013]

[18] J. A. Hendler. A little semantics goes a long way. [Online]. Available: http://www.cs.rpi.edu/∼hendler/LittleSemanticsWeb.html [Retrieved Feb. 8, 2013]

[19] A. Hogan, J. Z. Pan, A. Polleres, and S. Decker, "Template rule optimisations for distributed reasoning over 1 billion linked data triples," in *Proc. 9th Int. Semantic Web Conf.*, 2010, pp. 337–353.

[20] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, "Scalable distributed reasoning using MapReduce," in *Proc. 8th Int. Semantic Web Conf.*, 2009, pp. 634–649.

[21] J. Weaver and J. A. Hendler, "Parallel materialization of the finite RDFS closure for hundreds of millions of triples," in *Proc. 8th Int. Semantic Web Conf.*, 2009, pp. 682–697.

[22] M. Gilge. (2013, Feb. 11). *IBM System Blue Gene Solution: Blue Gene/Q Application Development* (2nd ed.). [Online]. Available: http://www.redbooks.ibm.com/redpieces/pdfs/sg247948.pdf [Retrieved Feb. 16, 2013]

[23] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semantics*, vol. 3, no. 2, pp. 158–182, Oct. 2005.

[24] A. Harth. Billion triple challenge 2012 dataset. [Online]. Available: http://km.aifb.kit.edu/projects/btc-2012/ [Retrieved Feb. 8, 2013]

[25] A. H. Karp and H. P. Flatt, "Measuring parallel processor performance," *Commun. ACM*, vol. 33, no. 5, pp. 539–543, May 1990.

[26] M. P. Bonacina, "A taxonomy of parallel strategies for deduction," *Ann. Math. and Artificial Intell.*, vol. 29, no. 1, pp. 223–257, Feb. 2000.

[27] S. Kotoulas, F. van Harmelen, and J. Weaver, "Knowledge representation and reasoning on the semantic web: Web-scale reasoning," in *Handbook of Semantic Web Technologies*, 1st ed., J. Domingue, D. Fensel, and J. A. Hendler, Eds. New York: Springer, 2011, ch. 11, pp. 441–466.

[28] C. L. Forgy, "OPS5 user's manual," Comput. Sci. Dept. Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. 2397, 1981.

[29] ——, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intell.*, vol. 19, no. 1, pp. 17–37, Sep. 1982.

[30] Y. Guo and J. Heflin. A scalable approach for partitioning OWL knowledge bases. presented at 2nd Int. Workshop Scalable Semantic Web Knowledge Base Systems. Athens, GA. [Online]. Available: http://swat.cse.lehigh.edu/pubs/guo06b.pdf [Retrieved Feb. 16, 2013]

[31] T. Liebig and F. Muller, "Parallelizing tableaux-based description logic reasoning," in *Proc. On the Move to Meaningful Internet Systems Workshop*, 2007, pp. 1135–1144.

[32] J. Bock, "Parallel computation techniques for ontology reasoning," in *Proc. 7th Int. Semantic Web Conf.*, 2008, pp. 901–906.

[33] E. L. Goodman and D. Mizell, "Scalable in-memory RDFS closure on billions of triples," in *Proc. 6th Int. Workshop Scalable Semantic Web Knowledge Base Systems*, 2010, pp. 17–31.

[34] S. Rudolph, T. Tserendorj, and P. Hitzler, "What is approximate reasoning?" in *Proc. Web Reasoning and Rule Systems Workshop*, 2008, pp. 150–164.

[35] R. Soma and V. K. Prasanna, "Parallel inferencing for OWL knowledge bases," in *Proc. 37th Int. Conf. Parallel Processing*, 2008, pp. 75–82.

[36] H. J. ter Horst, "Combining RDF and part of OWL with rules: Semantics, decidability, complexity," in *Proc. 4th Int. Semantic Web Conf.*, 2005, pp. 668–684.

[37] ——, "Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary," *J. Web Semantics*, vol. 3, no. 2, pp. 79–115, Oct. 2005.

[38] D. L. McGuinness. (2004, Feb.). OWL web ontology language overview. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2004/REC-owl-features-20040210/ [Retrieved Feb. 12, 2013]

[39] Z. Kaoudi, I. Miliaraki, and M. Koubarakis, "RDFS reasoning and query answering on top of DHTs," in *Proc. 8th Int. Semantic Web Conf.*, 2008, pp. 499–516.

[40] G. Anadiotis, S. Kotoulas, E. Oren, R. Siebes, F. van Harmelen, N. Drost, R. Kemp, J. Maassen, F. J. Seinstra, and H. E. Bal. MaRVIN: a distributed platform for massive RDF inference. presented at 7th Int. Semantic Web Conf. [Online]. Available: http://www.larkc.eu/marvin/btc2008.pdf [Retrieved Feb. 16, 2013]

[41] E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. ten Teije, and F. van Harmelen. (2009, Mar.). MaRVIN: A platform for large-scale analysis of semantic web data. WebSci'09: Society On-line. [Online]. Available: http://journal.webscience.org/232/2/websci09_submission_140.pdf [Retrieved Feb. 16, 2013]

[42] ——, "Marvin: Distributed reasoning over large-scale semantic web data," *J. Web Semantics*, vol. 7, no. 4, pp. 305–316, Dec. 2009.

[43] G. T. Williams, J. Weaver, M. Atre, and J. A. Hendler. Scalable reduction of large datasets to interesting subsets. presented at 8th Int. Semantic Web Conf. [Online]. Available: http://www.cs.rpi.edu/∼weavej3/papers/btc2009.pdf [Retrieved Feb. 25, 2013]

[44] ——, "Scalable Reduction of Large Datasets to Interesting Subsets," *J. Web Semantics*, vol. 8, no. 4, pp. 365–373, Nov. 2010.

[45] P. F. Patel-Schneider. (2012, Nov.). reasoning in RDFS is inherently serial, at least in the worst case. *Proc. ISWC 2012 Posters and Demonstrations Track*. [Online]. Available: http://ceur-ws.org/Vol-914/paper_1.pdf [Retrieved Feb. 16, 2013]

[46] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. Bal, "OWL reasoning with WebPIE: Calculating the closure of 100 billion triples," in *Proc. 7th Extended Semantic Web Conf.*, 2010, pp. 213–227.

[47] S. Kotoulas, E. Oren, and F. van Harmelen, "Mind the data skew: Distributed inferencing by speeddating in elastic regions," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 531–540.

[48] A. Hogan, A. Harth, and A. Polleres, "Scalable authoritative OWL reasoning for the web," *Int. J. Semantic Web and Inform. Syst.*, vol. 5, no. 2, pp. 49–90, 2009.

[49] C. Joslyn, B. Adolf, S. al-Saffar, J. Feo, E. Goodman, D. Haglin, G. Mackey, and D. Mizell. High performance semantic factoring of giga-scale semantic graph databases. presented at 9th Int. Semantic Web Conf. [Online]. Available: http://cass-mt.pnnl.gov/btc2010/pnnl_btc.pdf [Retrieved Feb. 16, 2013]

[50] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal, "QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases," in *Proc. 10th Int. Semantic Web Conf.*, 2011, pp. 730–745.

[51] J. Urbani, "On web-scale reasoning," Ph.D. dissertation, Comput. Sci. Dept., Vrije Universiteit, Amsterdam, Netherlands, 2013.

[52] N. Heino and J. Z. Pan, "RDFS reasoning on massively parallel hardware," presented at 11th Int. Semantic Web Conf., Boston, MA, 2012.

[53] J. Weaver. (2010, Jun.). Redefining the RDFS closure to be decidable. RDF Next Steps Workshop. [Online]. Available: http://www.w3.org/2009/12/rdf-ws/papers/ws16 [Retrieved Feb. 16, 2013]

[54] C. de Sainte Marie, G. Hallmark, and A. Paschke. (2010, Jun.). RIF production rule dialect. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/REC-rif-prd-20100622/ [Retrieved Feb. 12, 2013]

[55] M. Duerst and M. Suignard. (2005, Jan.). RFC 3987 – internationalized resource identifiers (IRIs). IETF. [Online]. Available: http://tools.ietf.org/html/rfc3987 [Retrieved Feb. 12, 2013]

[56] M. Birbeck and S. McCarron. (2010, Dec.). CURIE syntax 1.0. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/NOTE-curie-20101216/ [Retrieved Feb. 8, 2013]

[57] K. Knight, "Unification: A multidisciplinary survey," *ACM Computing Surveys*, vol. 21, no. 1, pp. 93–124, Mar. 1989.

[58] R. J. Bayardo, Jr. and R. C. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *Proc. 14th Nat. Conf. Artificial Intell. and 9th Conf. Innovative Applicat. of Artificial Intell.*, 1997, pp. 203–208.

[59] relsat - a propositional satisfiability solver and model counter. [Online]. Available: http://code.google.com/p/relsat/ [Retrieved Feb. 12, 2013]

[60] P. Hitzler and F. van Harmelen, "A reasonable semantic web," *Semantic Web J.*, vol. 1, no. 1, pp. 39–44, 2010.

[61] D. Reynolds. (2010, Jun.). OWL 2 RL in RIF. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/NOTE-rif-owl-rl-20100622/ [Retrieved Feb. 12, 2013]

[62] Message passing interface. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/ [Retrieved Feb. 12, 2013]

[63] D. Beckett. Redland RDF libraries. [Online]. Available: http://librdf.org/ [Retrieved Feb. 12, 2013]

[64] A. Polleres, "From SPARQL to rules (and back)," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 787–796.

[65] R. Hockney, "Performance parameters and benchmarking of supercomputers," *Parallel Computing*, vol. 17, no. 10, pp. 1111–1130, Dec. 1991.

[66] H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, and D. Reynolds. (2010, Jun.). RIF core dialect. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/REC-rif-core-20100622/ [Retrieved Feb. 12, 2013]

[67] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[68] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proc. Supercomputing '90*, 1990, pp. 324–333.

[69] A. Polleres, H. Boley, and M. Kifer. (2010, Jun.). RIF datatypes and built-ins 1.0. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2010/REC-rif-dtb-20100622/ [Retrieved Feb. 12, 2013]

[70] S. A. Cook, "A taxonomy of problems with fast parallel algorithms," *Inform. and Control*, vol. 64, no. 1, pp. 2–22, Jan. 1985.

[71] A. Hogan, A. Harth, and A. Polleres, "SAOR: Authoritative reasoning for the web," in *Proc. 3rd Asian Semantic Web Conf.*, 2008, pp. 76–90.

[72] J. Weaver and G. T. Williams. Reducing I/O load in parallel RDF systems via data compression. presented at 1st Workshop High-Performance Computing for the Semantic Web. Heraklion, Greece. [Online]. Available: http://ceur-ws.org/Vol-736/paper4.pdf [Retrieved Feb. 17, 2013]

[73] J. Weaver. Syntactic normalization of the BTC2012 dataset. [Online]. Available: http://tw.rpi.edu/weblog/2012/10/29/syntactic-normalization-of-the-btc2012-dataset/ [Retrieved Feb. 12, 2013]

[74] J. Weaver and G. T. Williams, "Scalable RDF query processing on clusters and supercomputers," in *Proc. 5th Int. Workshop Scalable Semantic Web Knowledge Base Systems*, 2009, pp. 73–85.

[75] J. Grant and D. Beckett. (2004, Feb.). RDF test cases. W3C. Cambridge, MA. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/ [Retrieved Feb. 12, 2013]

[76] R. Cyganiak, A. Harth, and A. Hogan. (2008, Jul.). N-quads: Extending N-triples with context. DERI. Galway, Ireland. [Online]. Available: http://sw.deri.org/2008/07/n-quads/ [Retrieved Feb. 12, 2013]

[77] Unicode 6.2.0. [Online]. Available: http://www.unicode.org/versions/Unicode6.2.0/ [Retrieved Feb. 17, 2013]

[78] A. Phillips and M. Davis. (2009, Sep.). RFC 5646 – tags for identifying languages. IETF. [Online]. Available: http://tools.ietf.org/html/rfc5646 [Retrieved Feb. 12, 2013]

[79] M. Davis and K. Whistler. (2012, Aug.). Unicode standard annex #15: Unicode normalization forms. Unicode Consortium. Mountain View, CA. [Online]. Available: http://www.unicode.org/reports/tr15/tr15-37.html [Retrieved Feb. 17, 2013]

[80] CCNI one of the world's most powerful university-based computing centers. [Online]. Available: http://ccni.rpi.edu/ [Retrieved Feb. 12, 2013]

[81] M. F. X. J. Oberhumer. oberhumer.com: LZO real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo/ [Retrieved Feb. 12, 2013]

[82] P. Konecny. Introducing the Cray XMT. presented at Cray User Group Meeting. [Online]. Available: https://cug.org/5-publications/proceedings_attendee_lists/2007CD/S07_Proceedings/pages/Authors/Konecny/Konecny_paper.pdf [Retrieved Feb. 17, 2013]

[83] J. Barkes, M. R. Barrios, F. Cougard, P. G. Crumley, D. Marin, H. Reddy, and T. Thitayanun. (1998, Apr.). *GPFS: A Parallel File System* (1st ed.). [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg245165.pdf [Retrieved Feb. 17, 2013]

[84] D. W. Bauer, Jr., C. D. Carothers, and A. Holder, "Scalable time warp on Blue Gene supercomputers," in *Proc. 2009 ACM/IEEE/SCS 23rd Workshop Principles of Advanced and Distributed Simulation*, 2009, pp. 35–44.

APPENDIX A
DICTIONARY ENCODING

As with all the code, the dictionary encoding code was written in C++, and MPI was used for interprocess communication. All (non-collective) communication and disk I/O was performed using asynchronous MPI calls. The dictionary encoding process begins by every processor collectively opening the (single) index file for the LZO-compressed data and using the offsets to divide the number of compressed blocks fairly evenly among processors. After this, the processors collectively open the (single) LZO-compressed N-triples file and begin reading from the beginning of their segments. Using the large block partition of the GPFS [83] file system at the CCNI, page sizes are 4 MB. All reads occur along page boundaries, reading one page at a time (except possibly in the cases of the beginning of the file segment and the end of the file segment, which may be partial pages). `MPI::File::Iread_at` is used for reading so that, which a processor was processing on page, the next page is being read. As pages are read, blocks in those pages are LZO-decompressed, and RDF triples are parsed. Each term in the RDF triple is independently dictionary encoded.

Skipping for a moment the actual dictionary encoding process, when a triple is finished being dictionary-encoded – that is, each term is replaced with a uniquely identifying 8-byte integer – the encoded triple is written into a four MB buffer. When the four MB buffer is full, it is written to an output file. Each processor opens its own output file in its own directory (to prevent contention over directory metadata) in the `MPI::COMM_SELF` communicator, and pages are written using the split-collective MPI routines `MPI::File::Write_at_all_begin` and `MPI::File::Write_at_all_end`. Using split-collective routines here is somewhat of an odd choice because the communicator in which the calls are being made is `MPI::COMM_SELF` which contains only the calling processor. Thus, there is really no need for a collective routine. These routines were used because they are part of a generic object for writing files in MPI that could be used with multiple processors

as well.

Regarding the dictionary encoding process, there are two distinct parts of the computation: the actual dictionary encoding, and the interprocess communication. The interprocess communication has three layers. From lowest level to highest level they are the packet distributor, the string distributor, and the controller.

A distributor (packet or string) has four main methods: `send`, `noMoreSends`, `receive`, and `done`. The `send` method takes a processor rank and a string of bytes and returns `true` if the bytes are *being sent* (not *already sent*) to the processor identified by the given rank. Otherwise, `false` is returned. Calling `noMoreSends` tells the distributor that `send` and `noMoreSends` will not be called again, and doing so would cause an exception to be thrown. Calling `receive` (no parameters) returns a string of bytes or `NULL`. `NULL` indicates that nothing could be immediately received, but *not* that there is nothing incoming. Finally, there is the `done` method which takes no parameters and returns a boolean value. The `done` method is a bit tricky because it is – by the interface definition of distributor – collective. That is, when one processor calls `done`, it should be expected that the processor will wait for all other processors to also call `done` before returning. Thus it is a potential source of deadlock. A processor should periodically call `done` even if it knows it is not done, just to ensure that other processors are not trapped in a call to `done`. If `done` returns `true`, then all processors have finished distribution, and no more calls should be made to distributor methods. `done` cannot return `true` until every processor has called `noMoreSends` and all the outstanding messages have been received.

The packet distributor uses timewarp-like communication with parameters similar to those in [84]. These parameters are referred to herein as the *packet size*, the *number of requests*, and the *coordination period*. The packet size is the fixed size of any given message to be sent. If `send` is called with a string of bytes of a different length, then an exception is thrown. In previous work [74], I had used variable-length messages, requiring a receiving process to first receive a message containing the size of the subsequent message, allocate enough space for the subsequent message, and then receive the subsequent message. This proved to significantly inhibit parallelism when there is frequent interprocessor communication,

and using fixed-size packets is intended to remedy that problem. The number of requests is the maximum number of asynchronous send requests *and* the maximum number of asynchronous receive requests that a processor can have. Let the number of requests be N. Upon initialization, the packet distributor immediately makes N asynchronous receive requests using `MPI::Comm::Irecv`. When `receive` is called, `MPI::Request::Testany` is used to check for a completed receive request. If one is found, the packet (as a string of bytes) is returned, and another `MPI::Comm::Irecv` is started. If no receive request has completed, then `NULL` is returned. Similarly, when `send` is called, `MPI::Request::Testany` is used to check for inactive or completed send requests. If one is found, then a new send request is started with `MPI::Comm::Isend`, and `true` is returned. Otherwise, `false` is returned.

The tricky part to the packet distributor is the handling of the call to `done`, and it is in this regard that the packet distributor is like distributed timewarp computations. The packet distributor keeps two counts: the number of calls to `done`, call it the "done count"; and the number of *net* messages, call it the "message count." The done count is initialized to zero, and the message count is initialized to one. Every time a send request is started, the message count is incremented, and every time a receive request is completed, the message count is decremented. Every time `done` is called, the done count is incremented. If the done count is less than the coordination period, then `done` immediately returns `false`. Otherwise, the done count is reset to zero, and the processor calls `MPI::Comm::Allreduce` to sum over the processors' message count. If the sum of the message counts is non-zero, `false` is returned. Otherwise, all receive requests are cancelled (there are always N outstanding) and `true` is returned. The important thing to understand here is how the sum of the processors' message counts equaling zero is an indication of having finished. When a processor calls `noMoreSends`, the message count is decremented to effectively undo the initialization to one. Therefore, letting p be the number of processors and s the sum of the message counts, upon initialization, $s = p$. Prior to any processor calling `noMoreSends`, $s = m + p$ where m is the number of messages that have been sent but not received (from the perspective of the packet distributor). Notice that even if all sent messages have been received $m = 0$, $s > 0$. This is

appropriate because a processor might still have more packets to send but just has yet to tell the packet distributor. However, when the processor calls `noMoreSends`, this tells the packet distributor that `send` will not be called again. Letting q be the number of processors that have called `noMoreSends`, then $s = m + p - q$. As long as $q < p$, it is impossible for $s = 0$. Finally, when all processors have called `noMoreSends`, $p = q$ and $s = m$. When $m = 0$, then truly, there are no more outstanding messages, and no processors will send any more messages. Therefore, distribution is finished.

The parameters to the packet distributor have significant impact on performance. From experience, for dictionary encoding, a packet size of 128 bytes seemed to work well (this may make more sense after the discussion of the string distributor). Consulting professor and distributed timewarp computation expert Chris Carothers concerning his experience with similar parameters when using a Blue Gene, he stated that the using eight and 4,096 for the timewarp parameters analogous to number of requests and coordination period had worked well for him. Starting with these values, neighboring values were also tested, none of which appeared to improve performance (and in some cases, worsened performance). Therefore, when using a packet distributor (which will be discussed again later), number of requests was always set to eight, and the coordination period was always set to 4,096. Clearly, these parameters are architecture dependent. One must be particularly careful with number of requests, coordination period, and the manner in which the `done` method is periodically called. Allowing too many send requests and a long coordination period can result in underlying MPI message buffers to quickly eat up memory. Even if the parameters are reasonably set, though, if the `done` method is not called with regularity on all processors, one processor can fall behind on receives or race ahead on sends, and the result will be the same.

The string distributor operates on top of a packet distributor and is responsible for relaying variable length messages. It breaks down messages into individual packets. Letting the packet size be n, if `send` is called with a message of length less than or equal to $n -$ `sizeof(size_t)`, then the length of the message is written into the first `sizeof(size_t)` bytes of a packet, and the message is written in

the remainder of the packet. The string distributor then calls `send` on the packet distributor with that packet. However, if the message to be sent is larger than $n -$ `sizeof(size_t)`, then the message is broken down into multiple packets. The packets will have a header of `sizeof(size_t)` + `sizeof(int)` + 8 bytes where the first `sizeof(size_t)` bytes stores the overall message length, the next `sizeof(int)` bytes identifies the sending processor, the following four bytes contains a message identifier, and the last four bytes contains the relative placement of the packet among the other packets for the same message. The string distributor then calls `send` on the underlying packet distributor to send all the packets. Recall, though, that the packet distributor can refuse to send a packet by returning `false`. In this case, the string distributor buffers the packets. Then `true` is returned. Later, when `send` or `done` is called, the string distributor attempts to send the packets again. In the case of `send`, if any of the buffered packets could not be sent, then `false` is returned. That is, the string distributor will only buffer one message at a time and will refuse other messages until it is able to send the buffered packets. If all the buffered packets are successfully sent (to the packet distributor), then the string distributor will attempt to send the new message as well.

Receiving messages in the string distributor is a bit like putting together a puzzle. A `std::multimap` is used to collect incoming packets. The keys of the `std::multimap` are the (treated as a unit) message length, sender rank, and message identifier. When `receive` is called, the string distributor calls `receive` on the underlying packet distributor for another packet. If no packet is available, the string distributor returns `NULL`. If a packet is received, the string distributor check to see if it is the last packet for a message, and if so, composes the message (it could be just the single packet) and returns it. Otherwise, it places the packet into the `std::multimap` and returns `false`.

Now when `noMoreSends` is called, the string distributor notes that is has been called, but it will not call `noMoreSends` on the underlying packet distributor if it has buffered packets. If it does have buffered packets, it will attempt to send them, and if they can all be sent, it will call `noMoreSends` on the packet distributor. When `done` is called, if there are buffered packets, the string distributor attempts to send

them. If it is successful, then it will call `noMoreSends` on the packet distributor. Regardless, `done` is called on the packet distributor and its value returned.

Having described the packet distributor which handles fixed-length messages, and having described the string distributor which handles variable-length messages, the final component to be discussed is the controller. The controller coordinates between a distributed computation and a distributor, and so a distributed computation must first be described.

A distributed computation has two main methods: `pickup` and `dropoff`. When `pickup` is called, it is a request by the controller that the distributed computation provide a message to be sent. If the distributed computation has a message to send, it writes it into the provided buffer (which may be resized as necessary) and returns the rank of the processor to which the message should be sent. Alternatively, it may return -1 to indicate there is currently nothing to send, or a value less than -1 to indicate that there is no more need to send messages. A call to `dropoff` simply provides a received message to the distributed computation.

The function of the controller is given in algorithm 4 which consists primarily of two loops. In the first loop, a message is "picked up" (if $sendto \geq 0$) and the inner "receiving" loop is entered. Every iteration of the receiving loop attempts to receive a message, and if a message is received, it is "dropped off" to the distribution computation. The repeating condition of this condition loop is important. Note that || is *short-circuited*, meaning that the subconditions are evaluation from left to right, and once a subcondition evaluates to true, the entire condition is considered true and no subconditions to the right are evaluated. The call too *dist.done* will always return `false` on line eight, but as mentioned, it is important to call it periodically to prevent deadlock, which is why it is placed in the loop condition. Then, if $sendto < 0$, then that means the distributed computation did not have a message to send, and so this will short-circuit the evaluation of the loop condition and prevent *dist.send* from being called. However, if $sendto \geq 0$, then *dist.send* will be called, and only when the send is successful will the receiving loop terminate. The first loop (lines 1-9) will terminate when the distributed computation indicates that there are no more messages to be sent, and $dist.noMoreSends$ is called on line 10. Lines

11-16 are another receiving loop which terminates when *dist.done* returns `true`, indicating that there are no more messages being sent and no more messages to be received.

Algorithm 4: Control between Distributed Computation and Distributor

```
   Input: Distributor dist and distributed computation comp.
   /* buffer and msg are resizable strings of bytes            */
   /* sendto is an integer                                     */
 1 repeat
 2 |   sendto = comp.pickup(buffer)
 3 |   repeat
 4 |   |   msg = dist.receive()
 5 |   |   if msg ≠ NULL then
 6 |   |   |   comp.dropoff(msg)
 7 |   |   end
 8 |   until dist.done() || sendto < 0 || dist.send(sendto, buffer)
 9 until sendto < -1
10 dist.noMoreSends()
11 repeat
12 |   msg = dist.receive()
13 |   if msg ≠ NULL then
14 |   |   comp.dropoff(msg)
15 |   end
16 until dist.done()
```

Finally, the actual dictionary encoding can be described as a distributed computation. Recall that triples are continually being read and encoded triples continually being buffered and written to disk. In between, they are being encoded. Algorithm 5 gives a high-level overview of the pickup method. On line 1, the `cached_response` method checks to see if there are any lookup responses cached that need to be sent (since the pickup method is the mechanism by which messages are sent), and if so, writes the lookup response in the buffer, writes the requesting processor's rank in `sendto`, and returns `true`. Otherwise, `cached_response` returns `false`. On line 4, the `next_lookup` method checks to see whether another lookup request needs to be made, and if so, sets `term` to the term for which lookup is needed and returns `true`. Otherwise, `next_lookup` returns `false`. On line 5, `check_termination` handles termination of the distributed dictionary encoding

computation, an important method which will be described in greater detail later. On line 7, the `lookup_in_progress` method checks to see if there is already an outstanding lookup request and, if so, caches `term` to await for completion of the request and returns `true`. Otherwise, `lookup_in_progress` returns `false`. On line 10, `local_lookup` checks to see whether the encoded value for `term` already exists in the local dictionary, and if so, encodes the term and returns `true`. Otherwise, `local_lookup` returns `false`. On line 13, `remote_lookup` determines which processor is responsible for encoding `term`, writes a lookup request in the buffer, and return the rank of the processor to which the request should be sent.

The dropoff method for distributed dictionary encoding is given in algorithm 6. On line 1, `message_is_no_more_requests` returns `true` iff the buffer contains a message stating that the sending processor will make no more requests. On line 4, `message_is_no_responses_expected` returns `true` iff the buffer contains a message stating that the sending processor expects no more responses. On line 7, `message_is_request` returns `true` iff buffer contains a lookup request.

Algorithm 5: Pickup Method for Distributed Dictionary Encoding

Input: A buffer *buffer* in which to write the message.
Output: The rank of the processor to which the message should be sent, or -1 to indicate no message, or less than -1 to indicate no more messages.

```
1  if cached_response(buffer, sendto) then
2  |  return sendto
3  end
4  if !next_lookup(term) then
5  |  return check_termination(buffer)
6  end
7  if lookup_in_progress(term) then
8  |  return -1
9  end
10 if local_lookup(term) then
11 |  return -1
12 end
13 return remote_lookup(buffer, term)
```

Algorithms 5 and 6 have been given to provide clarity concerning the overall flow of the distributed computation. Greater detail is not given in the form of

Algorithm 6: Dropoff Method for Distributed Dictionary Encoding

```
   Input: A buffer containing a message.
 1 if message_is_no_more_requests(buffer) then
 2 |   increment the "no more requests" counter
 3 end
 4 if message_is_no_responses_expected(buffer) then
 5 |   increment the "no responses expected" counter
 6 end
 7 if message_is_request(buffer) then
 8 |   perform the lookup request and cache the response for pickup
 9 else
   |   /* buffer contains a lookup response                    */
10 |   encode terms awaiting the response
11 |   write out any completely encoded triples
12 end
```

pseudocode since, due to complexity, it would not be significantly clearer than the actual source code. Instead, a written description of the important points follows.

It is easiest to understand from the perspective of the lifecycle of a triple, from being read, to getting encoded, to being written. A triple is read in the call to `next_lookup` (after requests have been made for the previous triple and assuming there are any triples left to be read), and immediately a "pending triple" is created. A pending triple is an array of three 8-byte integers along with a count called the "need" of the pending triple. The need is initialized to three because none of the terms in the triple have been encoded yet.

Now consider the lifecycle of a single term, from being retrieved from a triple, to getting encoded, to being written into a pending triple. This term is the `term` parameter (by reference) of `next_lookup`. Skipping over `lookup_in_progress` for a moment, on line 10 of algorithm 5, the processor checks its local dictionary to see if it already has an ID set for the term. If so, the processor writes that ID into the current pending triple at the correct position and decrements the need of the pending triple. If the need is now zero, then the pending triple is fully encoded and written to output.

However, if the term does not have an ID associated with it in the local dictionary, then a remote lookup must be performed on line 13 of algorithm 5. Each

processor keeps a count of the number of lookup requests it has sent out. When a remote lookup is performed, the current value of the count is used as the request ID, and the count is incremented. A "pending position" is then created, which consists of a pointer to the current pending triple and an integer for the position of the term being looked up (zero, one, or two). The request ID is then associated with the pending position in a `std::multimap`. A hash function is used to determine which processor is responsible for encoding the term.

Eventually, that processor receives the lookup request in a call to dropoff. Skipping over lines 1-6 of algorithm 6 for a moment, on line 7, the processor determines that the message is a lookup request, and on line 8, it encodes the term and caches the response to be sent in a later call to pickup. Eventually, a call to pickup for that processor will select that cached response on line 1 of algorithm 5 and send it back to the requesting processor.

The requesting processor will receive the response in a call to dropoff and will go down to line 10 of algorithm 6. The response includes with it the request ID which is used to lookup all the pending positions associated with it in the `std::multimap`. The pending positions each specify a position in a specific pending triple to which the encoded term ID should be written. The ID is written to that position in the pending triple, and the need for that pending triple is decremented. If the need is zero, then the pending triple has been completely encoded and written to output. The term is then associated with the ID in the processor's local dictionary.

Returning to `lookup_in_progress` on line 7 of algorithm 5, every time a request is sent, the term is associated with the request ID in a `std::map`. That way, when `lookup_in_progress` is called, it checks that `std::map` to see if a request is in progress for the current term. If so, a new pending position is created and associated with the request ID, thus avoiding the need for an additional lookup.

One final detail remains to be explained, and that is `check_termination`. Since distributed dictionary encoding is not just a matter of redistributing data (since every request is followed by a response), determining when encoding is finished is a delicate matter. `next_lookup` returns `false` only when no more triples can be read. The first p (where p is the number of processors) calls to `check_termination`

write a special message in the buffer that indicates that the processor will make no more requests. One such message is sent to each processor. Each processor counts the number of such messages it receives on line 2 of algorithm 6. Subsequent calls to `check_termination` return -1 until the processor has no more pending positions (i.e., all of its requests have been answered). Then, the next p calls to `check_termination` write a special message into the buffer indicating that the processor expects no more responses. One of such messages is sent to each processor, and each processor counts such messages on line 5 of algorithm 6. Subsequent calls to `check_termination` return -1 until the sum of the "no more requests" counter and the "no responses expected" counter is equal to $2p$, in which case -2 is returned guaranteeing that there are no more messages.

The complexity here is because even if a processor will make no more lookup requests, it might still service lookup requests and hence need to send responses. Thus any one processor is not necessarily done sending messages unless all the processors are done sending messages, which is why this explicit termination test is required.

APPENDIX B
RULESETS

This appendix contains original and restricted, RDFS and OWL2RL rulesets. In the restricted versions, special sets of terms are used to more concisely represent restrictions. Using the terminology of Hogan et al. [19], these sets are the set of "metaclasses" MC and the set of "metaproperties" MP. $MPT = MP \cup \{\texttt{rdf:type}\}$. To improve the brevity of patterns and restricted rules, the $\in$ and $\notin$ symbols will be used with MP, MPT, and MC to compress multiple restrictions, patterns, or rules into (syntactically) single restrictions, patterns, or rules (respectively).

B.1 RDFS-based Rulesets

This section provides some of the specific details regarding restriction of the RDFS ruleset into the Par-RDFS ruleset. Table B.1 contains the RDFS ruleset prior to restriction. Note that it does not contain the infinite number of axiomatic triples of the form `rdf:_`i`[rdf:type->rdfs:ContainerMembershipProperty]` where i is any positive integer. Also, literal generalization (rules lg and gl from [17]) has been excluded since, in the context of RIF inference, there is no need to create RDF blank nodes representing RDF literals.

In section 4.3.1, the restriction of the RDFS ruleset into the Par-RDFS ruleset is described. The forced variable assignments for patterns are given in table B.4. Table B.5 gives the rules that were *not* eliminated, referred to herein as the Par-RDFS ruleset. Comparing table B.5 with table B.1 reveals which rules were eliminated. Rules marked with an asterisk (*) in table B.5 are rules that resulted from a split during step 2. Table B.4 gives patterns such that, when facts matching the patterns are replicated, parallel Par-RDFS inference is correct.

Table B.1: The RDFS Ruleset

Rule ID	`If And(...)`	`Then Do(Assert(...))`
rdf1	`?u[?a->?y]`	`?a[rdf:type->rdf:Property]`
rdf2	`?u[?a->?l]`	`?l[rdf:type->rdf:XMLLiteral]`

	External(pred:is-literal-XMLLiteral(?l))	
rdfs1	?u[?a->?l] External(pred:is-literal-PlainLiteral(?l))	?l[rdf:type->rdfs:Literal]
rdfs2	?p[rdfs:domain->?c] ?x[?p->?y]	?x[rdf:type->?c]
rdfs3	?p[rdfs:range->?c] ?x[?p->?y]	?y[rdf:type->?c]
rdfs4a	?u[?a->?x]	?u[rdf:type->rdfs:Resource]
rdfs4b	?u[?a->?v]	?v[rdf:type->rdfs:Resource]
rdfs5	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p3]	?p1[rdfs:subPropertyOf->?p3]
rdfs6	?u[rdf:type->rdf:Property]	?u[rdfs:subPropertyOf->?u]
rdfs7	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y]	?x[?p2->?y]
rdfs8	?u[rdf:type->rdfs:Class]	?u[rdfs:subClassOf->rdfs:Resource]
rdfs9	?c1[rdfs:subClassOf->?c2] ?x[rdf:type->?c1]	?x[rdf:type->?c2]
rdfs10	?u[rdf:type->rdfs:Class]	?u[rdfs:subClassOf->?u]
rdfs11	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c3]	?c1[rdfs:subClassOf->?c3]
rdfs12	?u[rdf:type->rdfs:ContainerMembershipProperty]	?u[rdfs:subPropertyOf->rdfs:member]
rdfs13	?u[rdf:type->rdfs:Datatype]	?u[rdfs:subClassOf->rdfs:Literal]
rdfsax1		rdf:type[rdf:type->rdf:Property]
rdfsax2		rdf:subject[rdf:type->rdf:Property]
rdfsax3		rdf:predicate[rdf:type->rdf:Property]
rdfsax4		rdf:object[rdf:type->rdf:Property]
rdfsax5		rdf:first[rdf:type->rdf:Property]
rdfsax6		rdf:rest[rdf:type->rdf:Property]
rdfsax7		rdf:value[rdf:type->rdf:Property]
rdfsax8		rdf:nil[rdf:type->rdf:List]
rdfsax9		rdf:type[rdfs:domain->rdf:Property]
rdfsax10		rdfs:domain[rdfs:domain->rdf:Property]
rdfsax11		rdfs:range[rdfs:domain->rdf:Property]
rdfsax12		rdfs:subPropertyOf[rdfs:domain->rdf:Property]
rdfsax13		rdfs:subClassOf[rdfs:domain->rdfs:Class]
rdfsax14		rdf:subject[rdfs:domain->rdf:Statement]
rdfsax15		rdf:predicate[rdfs:domain->rdf:Statement]
rdfsax16		rdf:object[rdfs:domain->rdf:Statement]
rdfsax17		rdfs:member[rdfs:domain->rdfs:Resource]
rdfsax18		rdf:first[rdfs:domain->rdf:List]
rdfsax19		rdf:rest[rdfs:domain->rdf:List]
rdfsax20		rdfs:seeAlso[rdfs:domain->rdfs:Resource]
rdfsax21		rdfs:isDefinedBy[rdfs:domain->rdfs:Resource]

rdfsax22		rdfs:comment[rdfs:domain->rdfs:Resource]
rdfsax23		rdfs:label[rdfs:domain->rdfs:Resource]
rdfsax24		rdf:value[rdfs:domain->rdfs:Resource]
rdfsax25		rdf:type[rdfs:range->rdfs:Class]
rdfsax26		rdfs:domain[rdfs:range->rdfs:Class]
rdfsax27		rdfs:range[rdfs:range->rdfs:Class]
rdfsax28		rdfs:subPropertyOf[rdfs:range->rdf:Property]
rdfsax29		rdfs:subClassOf[rdfs:range->rdfs:Class]
rdfsax30		rdf:subject[rdfs:range->rdfs:Resource]
rdfsax31		rdf:predicate[rdfs:range->rdfs:Resource]
rdfsax32		rdf:object[rdfs:range->rdfs:Resource]
rdfsax33		rdfs:member[rdfs:range->rdfs:Resource]
rdfsax34		rdf:first[rdfs:range->rdfs:Resource]
rdfsax35		rdf:rest[rdfs:range->rdf:List]
rdfsax36		rdfs:seeAlso[rdfs:range->rdfs:Resource]
rdfsax37		rdfs:isDefinedBy[rdfs:range->rdfs:Resource]
rdfsax38		rdfs:comment[rdfs:range->rdfs:Literal]
rdfsax39		rdfs:label[rdfs:range->rdfs:Resource]
rdfsax40		rdf:value[rdfs:range->rdfs:Resource]
rdfsax41		rdf:Alt[rdfs:subClassOf->rdfs:Container]
rdfsax42		rdf:Bag[rdfs:subClassOf->rdfs:Container]
rdfsax43		rdf:Seq[rdfs:subClassOf->rdfs:Container]
rdfsax44		rdfs:ContainerMembershipProperty[rdfs:subClassOf->rdf:Property]
rdfsax45		rdfs:isDefinedby[rdfs:subPropertyOf->rdfs:seeAlso]
rdfsax46		rdf:XMLLiteral[rdf:type->rdfs:Datatype]
rdfsax47		rdf:XMLLiteral[rdfs:subClassOf->rdfs:Literal]
rdfsax48		rdfs:Datatype[rdfs:subClassOf->rdfs:Class]

Table B.2: The RDFS Metaclasses and Metaproperties

MC	*MP*
rdfs:Class	rdfs:domain
rdfs:Datatype	rdfs:range
rdfs:ContainerMembershipProperty	rdfs:subClassOf
	rdfs:subPropertyOf

Table B.3: Forced Assignments from Steps 1 and 3 of the Methodology applied to the RDFS ruleset

Replicate (α)	Arbitrary (ϵ)

And(?x1[?x2->?x3] ?x2 $\in MP$) And(?x1[rdf:type->?x3] ?x3 $\in MC$) And(External(pred:is-literal-XMLLiteral(?x1))) And(External(pred:is-literal-PlainLiteral(?x1)))	And(?x1[?x2->?x3] ?x2 $\notin MPT$) And(?x1[rdf:type->?x3] ?x3 $\notin MC$)

Table B.4: Replication Patterns for Correct, Parallel Par-RDFS Inference

Replicate (α)
And(?x1[?x2->?x3] ?x2 $\in MP$) And(?x1[rdf:type->?x3] ?x3 $\in MC$) And(External(pred:is-literal-XMLLiteral(?x1))) And(External(pred:is-literal-PlainLiteral(?x1)))

Table B.5: The Par-RDFS Ruleset

Rule ID	If And(...)	Then Do(Assert(...))
rdf1	?u[?a->?y]	?a[rdf:type->rdf:Property]
rdf2	?u[?a->?l] External(pred:is-literal-XMLLiteral(?l))	?l[rdf:type->rdf:XMLLiteral]
rdfs1	?u[?a->?l] External(pred:is-literal-PlainLiteral(?l))	?l[rdf:type->rdfs:Literal]
rdfs2*	?p[rdfs:domain->?c] ?x[?p->?y] ?c $\notin MC$	?x[rdf:type->?c]
rdfs3*	?p[rdfs:range->?c] ?x[?p->?y] ?c $\notin MC$	?y[rdf:type->?c]
rdfs4a	?u[?a->?x]	?u[rdf:type->rdfs:Resource]
rdfs4b	?u[?a->?v]	?v[rdf:type->rdfs:Resource]
rdfs5	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p3]	?p1[rdfs:subPropertyOf->?p3]
rdfs7a*	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y] ?p2 $\notin MPT$	?x[?p2->?y]
rdfs7b*	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y] ?y $\notin MC$	?x[?p2->?y]
rdfs8	?u[rdf:type->rdfs:Class]	?u[rdfs:subClassOf->rdfs:Resource]
rdfs9*	?c1[rdfs:subClassOf->?c2] ?x[rdf:type->?c1] ?c2 $\notin MC$	?x[rdf:type->?c2]
rdfs10	?u[rdf:type->rdfs:Class]	?u[rdfs:subClassOf->?u]
rdfs11	?c1[rdfs:subClassOf->?c2]	?c1[rdfs:subClassOf->?c3]

	?c2[rdfs:subClassOf->?c3]	
rdfs12	?u[rdf:type->rdfs:ContainerMembershipProperty]	?u[rdfs:subPropertyOf->rdfs:member]
rdfs13	?u[rdf:type->rdfs:Datatype]	?u[rdfs:subClassOf->rdfs:Literal]
rdfsax1		rdf:type[rdf:type->rdf:Property]
rdfsax2		rdf:subject[rdf:type->rdf:Property]
rdfsax3		rdf:predicate[rdf:type->rdf:Property]
rdfsax4		rdf:object[rdf:type->rdf:Property]
rdfsax5		rdf:first[rdf:type->rdf:Property]
rdfsax6		rdf:rest[rdf:type->rdf:Property]
rdfsax7		rdf:value[rdf:type->rdf:Property]
rdfsax8		rdf:nil[rdf:type->rdf:List]
rdfsax9		rdf:type[rdfs:domain->rdf:Property]
rdfsax10		rdfs:domain[rdfs:domain->rdf:Property]
rdfsax11		rdfs:range[rdfs:domain->rdf:Property]
rdfsax12		rdfs:subPropertyOf[rdfs:domain->rdf:Property]
rdfsax13		rdfs:subClassOf[rdfs:domain->rdfs:Class]
rdfsax14		rdf:subject[rdfs:domain->rdf:Statement]
rdfsax15		rdf:predicate[rdfs:domain->rdf:Statement]
rdfsax16		rdf:object[rdfs:domain->rdf:Statement]
rdfsax17		rdfs:member[rdfs:domain->rdfs:Resource]
rdfsax18		rdf:first[rdfs:domain->rdf:List]
rdfsax19		rdf:rest[rdfs:domain->rdf:List]
rdfsax20		rdfs:seeAlso[rdfs:domain->rdfs:Resource]
rdfsax21		rdfs:isDefinedBy[rdfs:domain->rdfs:Resource]
rdfsax22		rdfs:comment[rdfs:domain->rdfs:Resource]
rdfsax23		rdfs:label[rdfs:domain->rdfs:Resource]
rdfsax24		rdf:value[rdfs:domain->rdfs:Resource]
rdfsax25		rdf:type[rdfs:range->rdfs:Class]
rdfsax26		rdfs:domain[rdfs:range->rdfs:Class]
rdfsax27		rdfs:range[rdfs:range->rdfs:Class]
rdfsax28		rdfs:subPropertyOf[rdfs:range->rdf:Property]
rdfsax29		rdfs:subClassOf[rdfs:range->rdfs:Class]
rdfsax30		rdf:subject[rdfs:range->rdfs:Resource]
rdfsax31		rdf:predicate[rdfs:range->rdfs:Resource]
rdfsax32		rdf:object[rdfs:range->rdfs:Resource]
rdfsax33		rdfs:member[rdfs:range->rdfs:Resource]
rdfsax34		rdf:first[rdfs:range->rdfs:Resource]
rdfsax35		rdf:rest[rdfs:range->rdf:List]
rdfsax36		rdfs:seeAlso[rdfs:range->rdfs:Resource]
rdfsax37		rdfs:isDefinedBy[rdfs:range->rdfs:Resource]
rdfsax38		rdfs:comment[rdfs:range->rdfs:Literal]
rdfsax39		rdfs:label[rdfs:range->rdfs:Resource]

rdfsax40		rdf:value[rdfs:range->rdfs:Resource]
rdfsax41		rdf:Alt[rdfs:subClassOf->rdfs:Container]
rdfsax42		rdf:Bag[rdfs:subClassOf->rdfs:Container]
rdfsax43		rdf:Seq[rdfs:subClassOf->rdfs:Container]
rdfsax44		rdfs:ContainerMembershipProperty[rdfs:subClassOf->rdf:Property]
rdfsax45		rdfs:isDefinedby[rdfs:subPropertyOf->rdfs:seeAlso]
rdfsax46		rdf:XMLLiteral[rdf:type->rdfs:Datatype]
rdfsax47		rdf:XMLLiteral[rdfs:subClassOf->rdfs:Literal]
rdfsax48		rdfs:Datatype[rdfs:subClassOf->rdfs:Class]

B.2 OWL2-based Rulesets

This section provides some of the specific details regarding restriction of the OWL2RL ruleset into the Par-OWL2 ruleset. Table B.6 contains the OWL2RL ruleset prior to restriction. Note that the OWL2RL rules presented herein are different than those from [3]. They are a RIF variation of the OWL2RL rules from [61], deviating to make the rules amenable to forward-chaining, following advice from [61] as well.

In section 4.3.2, the restriction of the OWL2 ruleset into the Par-OWL2 ruleset is briefly described. The forced variable assignments for patterns are given in table B.8. Table B.10 gives the rules that were *not* eliminated, referred to herein as the Par-OWL2 ruleset. Comparing table B.10 with table B.6 reveals which rules were eliminated. Rules marked with an asterisk (*) in table B.10 are rules that resulted from a split during step 2. Table B.8 gives patterns such that, when facts matching the patterns are replicated, parallel Par-OWL2 inference is correct.

Rules marked with a dagger (†) in table B.10 are rules that were excluded in the evaluation. The ruleset consisting of the Par-OWL2 rules *not* marked with a dagger is referred to as the Par-MemOWL2 ruleset. Discussion regarding the reason for further restriction is given in section 5.1.4.

Table B.6: The OWL2RL Ruleset

scm-int	?c[owl:intersectionOf->?l]	_markAllTypes(?c ?l)
scm-int-1	_markAllTypes(?c ?r)	_markAllTypes(?c ?l)

	?r[rdf:rest->?l] Not(?l = rdf:nil)	
scm-int-2	_markAllTypes(?c ?l) ?l[rdf:first->?ci]	?c[rdfs:subClassOf->?ci]
scm-uni	?c[owl:unionOf->?l]	_checkUnionOf(?c ?l)
scm-uni-1	_checkUnionOf(?c ?r) ?r[rdf:rest->?l] Not(?l = rdf:nil)	_checkUnionOf(?c ?l)
scm-uni-2	_checkUnionOf(?c ?l) ?l[rdf:first->?ci]	?ci[rdfs:subClassOf->?c]
scm-cls	?c[rdf:type->owl:Class]	?c[rdfs:subClassOf->?c]
scm-cls1	?c[rdf:type->owl:Class]	?c[owl:equivalentClass->?c]
scm-cls2	?c[rdf:type->owl:Class]	?c[rdfs:subClassOf->owl:Thing]
scm-cls3	?c[rdf:type->owl:Class]	owl:Nothing[rdfs:subClassOf->?c]
scm-sco	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c3]	?c1[rdfs:subClassOf->?c3]
scm-eqc1	?c1[owl:equivalentClass->?c2]	?c1[rdfs:subClassOf->?c2]
scm-eqc11	?c1[owl:equivalentClass->?c2]	?c2[rdfs:subClassOf->?c1]
scm-eqc2	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c1]	?c1[owl:equivalentClass->?c2]
scm-op	?p[rdf:type->owl:ObjectProperty]	?p[rdfs:subPropertyOf->?p]
scm-op1	?p[rdf:type->owl:ObjectProperty]	?p[owl:equivalentProperty->?p]
scm-dp	?p[rdf:type->owl:DatatypeProperty]	?p[rdfs:subPropertyOf->?p]
scm-dp1	?p[rdf:type->owl:DatatypeProperty]	?p[owl:equivalentProperty->?p]
scm-spo	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p3]	?p1[rdfs:subPropertyOf->?p3]
scm-eqp1	?p1[owl:equivalentProperty->?p2]	?p1[rdfs:subPropertyOf->?p2]
scm-eqp11	?p1[owl:equivalentProperty->?p2]	?p2[rdfs:subPropertyOf->?p1]
scm-eqp2	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p1]	?p1[owl:equivalentProperty->?p2]
scm-dom1	?p[rdfs:domain->?c1] ?c1[rdfs:subClassOf->?c2]	?p[rdfs:domain->?c2]
scm-dom2	?p2[rdfs:domain->?c] ?p1[rdfs:subPropertyOf->?p2]	?p1[rdfs:domain->?c]
scm-rng1	?p[rdfs:range->?c1] ?c1[rdfs:subClassOf->?c2]	?p[rdfs:range->?c2]
scm-rng2	?p2[rdfs:range->?c] ?p1[rdfs:subPropertyOf->?p2]	?p1[rdfs:range->?c]
scm-hv	?c1[owl:hasValue->?i] ?c1[owl:onProperty->?p1] ?c2[owl:hasValue->?i] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c1[rdfs:subClassOf->?c2]
scm-svf1	?c1[owl:someValuesFrom->?y1] ?c1[owl:onProperty->?p] ?c2[owl:someValuesFrom->?y2] ?c2[owl:onProperty->?p]	?c1[rdfs:subClassOf->?c2]

	?y1[rdfs:subClassOf->?y2]	
scm-svf2	?c1[owl:someValuesFrom->?y] ?c1[owl:onProperty->?p1] ?c2[owl:someValuesFrom->?y] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c1[rdfs:subClassOf->?c2]
scm-avf1	?c1[owl:allValuesFrom->?y1] ?c1[owl:onProperty->?p] ?c2[owl:allValuesFrom->?y2] ?c2[owl:onProperty->?p] ?y1[rdfs:subClassOf->?y2]	?c1[rdfs:subClassOf->?c2]
scm-avf2	?c1[owl:allValuesFrom->?y] ?c1[owl:onProperty->?p1] ?c2[owl:allValuesFrom->?y] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c2[rdfs:subClassOf->?c1]
eq-ref	?s[?p->?o]	?s[owl:sameAs->?s]
eq-ref1	?s[?p->?o]	?p[owl:sameAs->?p]
eq-ref2	?s[?p->?o]	?o[owl:sameAs->?o]
eq-sym	?x[owl:sameAs->?y]	?y[owl:sameAs->?x]
eq-trans	?x[owl:sameAs->?y] ?y[owl:sameAs->?z]	?x[owl:sameAs->?z]
eq-rep-s	?s[owl:sameAs->?s2] ?s[?p->?o]	?s2[?p->?o]
eq-rep-p	?p[owl:sameAs->?p2] ?s[?p->?o]	?s[?p2->?o]
eq-rep-o	?o[owl:sameAs->?o2] ?s[?p->?o]	?s[?p->?o2]
eq-diff1	?x[owl:sameAs->?y] ?x[owl:differentFrom->?y]	rif:error()
prp-ap-l		rdfs:label[rdf:type->owl:AnnotationProperty]
prp-ap-c		rdfs:comment[rdf:type->] owl:AnnotationProperty]
prp-ap-sa		rdfs:seeAlso[rdf:type->] owl:AnnotationProperty]
prp-ap-idb		rdfs:isDefinedBy[rdf:type->] owl:AnnotationProperty]
prp-ap-d		owl:deprecated[rdf:type->] owl:AnnotationProperty]
prp-ap-pv		owl:priorVersion[rdf:type->] owl:AnnotationProperty]
prp-ap-bcw		owl:backwardCompatibleWith[rdf:type->] owl:AnnotationProperty]
prp-ap-iw		owl:incompatibleWith[rdf:type->] owl:AnnotationProperty]
prp-dom	?p[rdfs:domain->?c]	?x[rdf:type->?c]

	?x[?p->?y]	
prp-rng	?p[rdfs:range->?c] ?x[?p->?y]	?y[rdf:type->?c]
prp-fp	?p[rdf:type->owl:FunctionalProperty] ?x[?p->?y1] ?x[?p->?y2]	?y1[owl:sameAs->?y2]
prp-ifp	?p[rdf:type->owl:InverseFunctionalProperty] ?x1[?p->?y] ?x2[?p->?y]	?x1[owl:sameAs->?x2]
prp-irp	?p[rdf:type->owl:IrreflexiveProperty] ?x[?p->?x]	rif:error()
prp-symp	?p[rdf:type->owl:SymmetricProperty] ?x[?p->?y]	?y[?p->?x]
prp-asyp	?p[rdf:type->owl:AsymmetricProperty] ?x[?p->?y] ?y[?p->?x]	rif:error()
prp-trp	?p[rdf:type->owl:TransitiveProperty] ?x[?p->?y] ?y[?p->?z]	?x[?p->?z]
prp-spo1	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y]	?x[?p2->?y]
prp-eqp1	?p1[owl:equivalentProperty->?p2] ?x[?p1->?y]	?x[?p2->?y]
prp-eqp2	?p1[owl:equivalentProperty->?p2] ?x[?p2->?y]	?x[?p1->?y]
prp-pdw	?p1[owl:propertyDisjointWith->?p2] ?x[?p1->?y] ?x[?p2->?y]	rif:error()
prp-inv1	?p1[owl:inverseOf->?p2] ?x[?p1->?y]	?y[?p2->?x]
prp-inv2	?p1[owl:inverseOf->?p2] ?x[?p2->?y]	?y[?p1->?x]
cls-thing		owl:Thing[rdf:type->owl:Class]
cls-nothing1		owl:Nothing[rdf:type->owl:Class]
cls-nothing2	?x[rdf:type->owl:Nothing]	rif:error()
cls-svf1	?x[owl:someValuesFrom->?y] ?x[owl:onProperty->?p] ?u[?p->?v] ?v[rdf:type->?y]	?u[rdf:type->?x]
cls-svf2	?x[owl:someValuesFrom->owl:Thing] ?x[owl:onProperty->?p] ?u[?p->?v]	?u[rdf:type->?x]
cls-avf	?x[owl:allValuesFrom->?y] ?x[owl:onProperty->?p] ?u[rdf:type->?x] ?u[?p->?v]	?v[rdf:type->?y]
cls-hv1	?x[owl:hasValue->?y]	?u[?p->?y]

	?x[owl:onProperty->?p] ?u[rdf:type->?x]	
cls-hv2	?x[owl:hasValue->?y] ?x[owl:onProperty->?p] ?u[?p->?y]	?u[rdf:type->?x]
cls-maxc1	?x[owl:maxCardinality->0] ?x[owl:onProperty->?p] ?u[rdf:type->?x] ?u[?p->?y]	rif:error()
cls-maxc2	?x[owl:maxCardinality->1] ?x[owl:onProperty->?p] ?u[rdf:type->?x] ?u[?p->?y1] ?u[?p->?y2]	?y1[owl:sameAs->?y2]
cls-maxqc1	?x[owl:maxQualifiedCardinality->0] ?x[owl:onProperty->?p] ?x[owl:onClass->?c] ?u[rdf:type->?x] ?u[?p->?y] ?y[rdf:type->?c]	rif:error()
cls-maxqc2	?x[owl:maxQualifiedCardinality->0] ?x[owl:onProperty->?p] ?x[owl:onClass->owl:Thing] ?u[rdf:type->?x] ?u[?p->?y]	rif:error()
cls-maxqc3	?x[owl:maxQualifiedCardinality->1] ?x[owl:onProperty->?p] ?x[owl:onClass->?c] ?u[rdf:type->?x] ?u[?p->?y1] ?y1[rdf:type->?c] ?u[?p->?y2] ?y2[rdf:type->?c]	?y1[owl:sameAs->?y2]
cls-maxqc4	?x[owl:maxQualifiedCardinality->1] ?x[owl:onProperty->?p] ?x[owl:onClass->owl:Thing] ?u[rdf:type->?x] ?u[?p->?y1] ?u[?p->?y2]	?y1[owl:sameAs->?y2]
cax-sco	?c1[rdfs:subClassOf->?c2] ?x[rdf:type->?c1]	?x[rdf:type->?c2]
cax-eqc1	?c1[owl:equivalentClass->?c2] ?x[rdf:type->?c1]	?x[rdf:type->?c2]
cax-eqc2	?c1[owl:equivalentClass->?c2] ?x[rdf:type->?c2]	?x[rdf:type->?c1]
cax-dw	?c1[owl:disjointWith->?c2] ?x[rdf:type->?c1]	rif:error()

	?x[rdf:type->?c2]	
prp-npa1	?x[owl:sourceIndividual->?i1] ?x[owl:assertionProperty->?p] ?x[owl:targetIndividual->?i2] ?i1[?p->?i2]	rif:error()
prp-npa2	?x[owl:sourceIndividual->?i] ?x[owl:assertionProperty->?p] ?x[owl:targetValue->?lt] ?i[?p->?lt]	rif:error()
cax-dw	?c1[owl:disjointWith->?c2] ?x[rdf:type->?c1] ?x[rdf:type->?c2]	rif:error()
cls-com	?c1[owl:complementOf->?c2] ?x[rdf:type->?c1] ?x[rdf:type->?c2]	rif:error()
eq-diff2a	?x[rdf:type->owl:AllDifferent] ?x[owl:distinctMembers->?y]	_checkDifferent(?x ?y)
eq-diff3a	?x[rdf:type->owl:AllDifferent] ?x[owl:members->?y]	_checkDifferent(?x ?y)
eq-diff23b	_checkDifferent(?x ?z) ?z[rdf:rest->?y] Not(?y = rdf:nil)	_checkDifferent(?x ?y)
eq-diff23c	_checkDifferent(?x ?y1) _checkDifferent(?x ?y2) Not(?y1 = ?y2) ?y1[rdf:first->?z1] ?y2[rdf:first->?z2] ?z1[owl:sameAs->?z2]	rif:error()
prp-adp	?r[rdf:type->owl:AllDisjointProperties] ?r[owl:members -> ?l]	_checkDisjointProperties(?r ?l)
prp-adp-1	_checkDisjointProperties(?r ?x) ?x[rdf:rest->?l] Not(?l = rdf:nil)	_checkDisjointProperties(?r ?l)
prp-adp-2	_checkDisjointProperties(?r ?l1) _checkDisjointProperties(?r ?l2) Not(?l1 = ?l2) ?l1[rdf:first->?x] ?l2[rdf:first->?y] ?o[?x->?v] ?o[?y->?v]	rif:error()
cax-adc	?r[rdf:type -> owl:AllDisjointClasses] ?r[owl:members -> ?l]	_checkDisjointClasses(?r ?l)
cax-adc-1	_checkDisjointClasses(?r ?x) ?x[rdf:rest->?l] Not(?l = rdf:nil)	_checkDisjointClasses(?r ?l)
cax-adc-2	_checkDisjointClasses(?r ?l1) _checkDisjointClasses(?r ?l2)	rif:error()

	Not(?l1 = ?l2) ?l1[rdf:first->?x] ?l2[rdf:first->?y] ?o[rdf:type->?x] ?o[rdf:type->?y]	
prp-spo2	?p[owl:propertyChainAxiom->?pc]	_markCheckChain(?p ?pc)
prp-spo2-1	_markCheckChain(?p ?q) ?q[rdf:rest->?pc] Not(?pc = rdf:nil)	_markCheckChain(?p ?pc)
prp-spo2-2	_markCheckChain(?q ?pc) ?pc[rdf:first->?p] ?pc[rdf:rest->rdf:nil] ?start[?p->?last]	_checkChain(?q ?start ?pc ?last)
prp-spo2-3	?pc[rdf:first->?p] ?pc[rdf:rest->?tl] ?start[?p->?next] _checkChain(?q ?next ?tl ?last)	_checkChain(?q ?start ?pc ?last)
prp-spo2-4	?p[owl:propertyChainAxiom->?pc] _checkChain(?p ?start ?pc ?last)	?start[?p->?last]
cls-int1	_markAllTypes(?c ?l) ?l[rdf:first->?ty] ?l[rdf:rest->rdf:nil] ?y[rdf:type->?ty]	_allTypes(?c ?l ?y)
cls-int1-1	?l[rdf:first->?ty] ?l[rdf:rest->?tl] ?y[rdf:type->?ty] _allTypes(?c ?tl ?y)	_allTypes(?c ?l ?y)
cls-int1-2	?c[owl:intersectionOf->?l] _allTypes(?c ?l ?y)	?y[rdf:type->?c]
prp-key	?c[owl:hasKey->?u]	_markSameKey(?c ?u)
prp-key-1	_markSameKey(?c ?v) ?v[rdf:rest->?u] Not(?u = rdf:nil)	_markSameKey(?c ?u)
prp-key-2	_markSameKey(?c ?u) ?u[rdf:first->?key] ?u[rdf:rest->rdf:nil] ?x[?key->?v] ?y[?key->?v]	_sameKey(?c ?u ?x ?y)
prp-key-3	?u[rdf:first->?key] ?u[rdf:rest->?tl] ?x[?key->?v] ?y[?key->?v] _sameKey(?c ?tl ?x ?y)	_sameKey(?c ?u ?x ?y)
prp-key-4	?c[owl:hasKey->?u] ?x[rdf:type->?c] ?y[rdf:type->?c] _sameKey(?c ?u ?x ?y)	?x[owl:sameAs->?y]

cls-uni	_checkUnionOf(?c ?l) ?l[rdf:first->?ci] ?y[rdf:type->?ci]	?y[rdf:type->?c]
cls-oo-a	?c[owl:oneOf->?l]	_checkOneOf(?c ?l)
cls-oo-b	_checkOneOf(?c ?r) ?r[rdf:rest->?l] Not(?l = rdf:nil)	_checkOneOf(?c ?l)
cls-oo-c	_checkOneOf(?c ?l) ?l[rdf:first->?yi]	?yi[rdf:type->?c]
cls-int2	_markAllTypes(?c ?l) ?l[rdf:first->?ci] ?y[rdf:type->?c]	?y[rdf:type->?ci]

Table B.7: The OWL2RL Metaclasses and Metaproperties

MC	*MP*
owl:FunctionalProperty	rdfs:domain
owl:InverseFunctionalProperty	rdfs:range
owl:IrreflexiveProperty	rdfs:subPropertyOf
owl:SymmetricProperty	owl:equivalentProperty
owl:AsymmetricProperty	owl:propertyDisjointWith
owl:TransitiveProperty	owl:inverseOf
owl:Class	owl:someValuesFrom
owl:ObjectProperty	owl:onProperty
owl:DatatypeProperty	owl:allValuesFrom
owl:AllDifferent	owl:hasValue
owl:AllDisjointProperties	owl:onClass
owl:AllDisjointClasses	rdfs:subClassOf
	owl:equivalentClass
	owl:disjointWith
	owl:complementOf
	owl:distinctMembers
	owl:members
	owl:propertyChainAxiom
	owl:intersectionOf
	owl:hasKey
	owl:unionOf
	owl:oneOf
	rdf:first
	rdf:rest
	owl:maxCardinality
	owl:maxQualifiedCardinality

Table B.8: Forced Assignments from Steps 1 and 3 of the Methodology applied to the OWL2RL ruleset

Replicate (α)	Arbitrary (ϵ)
And(?x1[?x2->?x3] ?x2 $\in MP$)	And(?x1[?x2->?x3] ?x2 $\notin MPT$)
And(?x1[rdf:type->?x3] ?x3 $\in MC$)	And(?x1[rdf:type->?x3] ?x3 $\notin MC$)
And(_markAllTypes(?x1 ?x2))	
And(_allTypes(?x1 ?x2))	
And(_checkUnionOf(?x1 ?x2))	
And(_checkDifferent(?x1 ?x2))	
And(_checkDisjointProperties(?x1 ?x2))	
And(_checkDisjointClasses(?x1 ?x2))	
And(_markCheckChain(?x1 ?x2))	
And(_checkChain(?x1 ?x2 ?x3 ?x4))	
And(_markSameKey(?x1 ?x2))	
And(_sameKey(?x1 ?x2 ?x3 ?x4))	
And(?x1 = ?x2)	

Table B.9: Replication Patterns for Correct, Parallel Par-OWL2 Inference

Replicate (α)
And(?x1[?x2->?x3] ?x2 $\in MP$)
And(?x1[rdf:type->?x3] ?x3 $\in MC$)
And(_markAllTypes(?x1 ?x2))
And(_allTypes(?x1 ?x2))
And(_checkUnionOf(?x1 ?x2))
And(_checkDifferent(?x1 ?x2))
And(_checkDisjointProperties(?x1 ?x2))
And(_checkDisjointClasses(?x1 ?x2))
And(_markCheckChain(?x1 ?x2))
And(_checkChain(?x1 ?x2 ?x3 ?x4))
And(_markSameKey(?x1 ?x2))
And(_sameKey(?x1 ?x2 ?x3 ?x4))
And(?x1 = ?x2)
And(_checkOneOf(?x1 ?x2))

Table B.10: The Par-OWL2 Ruleset

Rule ID	If And(...)	Then Do(Assert(...))
scm-int1	?c[owl:intersectionOf->?l]	_markAllTypes(?c ?l)
scm-int2	_markAllTypes(?c ?r) ?r[rdf:rest->?l] Not(?l = rdf:nil)	_markAllTypes(?c ?l)

scm-int3	_markAllTypes(?c ?l) ?l[rdf:first->?ci]	?c[rdfs:subClassOf->?ci]
scm-uni1	?c[owl:unionOf->?l]	_checkUnionOf(?c ?l)
scm-uni2	_checkUnionOf(?c ?r) ?r[rdf:rest->?l] Not(?l = rdf:nil)	_checkUnionOf(?c ?l)
scm-uni3	_checkUnionOf(?c ?l) ?l[rdf:first->?ci]	?ci[rdfs:subClassOf->?c]
scm-cls†	?c[rdf:type->owl:Class]	?c[rdfs:subClassOf->?c]
scm-cls1†	?c[rdf:type->owl:Class]	?c[owl:equivalentClass->?c]
scm-cls2†	?c[rdf:type->owl:Class]	?c[rdfs:subClassOf->owl:Thing]
scm-cls3†	?c[rdf:type->owl:Class]	owl:Nothing[rdfs:subClassOf->?c]
scm-sco	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c3]	?c1[rdfs:subClassOf->?c3]
scm-eqc1	?c1[owl:equivalentClass->?c2]	?c1[rdfs:subClassOf->?c2]
scm-eqc11	?c1[owl:equivalentClass->?c2]	?c2[rdfs:subClassOf->?c1]
scm-eqc2	?c1[rdfs:subClassOf->?c2] ?c2[rdfs:subClassOf->?c1]	?c1[owl:equivalentClass->?c2]
scm-op†	?p[rdf:type->owl:ObjectProperty]	?p[rdfs:subPropertyOf->?p]
scm-op1†	?p[rdf:type->owl:ObjectProperty]	?p[owl:equivalentProperty->?p]
scm-dp†	?p[rdf:type->owl:DatatypeProperty]	?p[rdfs:subPropertyOf->?p]
scm-dp1†	?p[rdf:type->owl:DatatypeProperty]	?p[owl:equivalentProperty->?p]
scm-spo	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p3]	?p1[rdfs:subPropertyOf->?p3]
scm-eqp1	?p1[owl:equivalentProperty->?p2]	?p1[rdfs:subPropertyOf->?p2]
scm-eqp11	?p1[owl:equivalentProperty->?p2]	?p2[rdfs:subPropertyOf->?p1]
scm-eqp2	?p1[rdfs:subPropertyOf->?p2] ?p2[rdfs:subPropertyOf->?p1]	?p1[owl:equivalentProperty->?p2]
scm-dom1	?p[rdfs:domain->?c1] ?c1[rdfs:subClassOf->?c2]	?p[rdfs:domain->?c2]
scm-dom2	?p2[rdfs:domain->?c] ?p1[rdfs:subPropertyOf->?p2]	?p1[rdfs:domain->?c]
scm-rng1	?p[rdfs:range->?c1] ?c1[rdfs:subClassOf->?c2]	?p[rdfs:range->?c2]
scm-rng2	?p2[rdfs:range->?c] ?p1[rdfs:subPropertyOf->?p2]	?p1[rdfs:range->?c]
scm-hv	?c1[owl:hasValue->?i] ?c1[owl:onProperty->?p1] ?c2[owl:hasValue->?i] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c1[rdfs:subClassOf->?c2]
scm-svf1	?c1[owl:someValuesFrom->?y1] ?c1[owl:onProperty->?p] ?c2[owl:someValuesFrom->?y2] ?c2[owl:onProperty->?p] ?y1[rdfs:sbuClassOf->?y2]	?c1[rdfs:subClassOf->?c2]

scm-svf2	?c1[owl:someValuesFrom->?y] ?c1[owl:onProperty->?p1] ?c2[owl:someValuesFrom->?y] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c1[rdfs:subClassOf->?c2]
scm-avf1	?c1[owl:allValuesFrom->?y1] ?c1[owl:onProperty->?p] ?c2[owl:allValuesFrom->?y2] ?c2[owl:onProperty->?p] ?y1[rdfs:subClassOf->?y2]	?c1[rdfs:subClassOf->?c2]
scm-avf2	?c1[owl:allValuesFrom->?y] ?c1[owl:onProperty->?p1] ?c2[owl:allValuesFrom->?y] ?c2[owl:onProperty->?p2] ?p1[rdfs:subPropertyOf->?p2]	?c2[rdfs:subClassOf->?c1]
eq-ref†	?s[?p->?o]	?s[owl:sameAs->?s]
eq-ref1†	?s[?p->?o]	?p[owl:sameAs->?p]
eq-ref2†	?s[?p->?o]	?o[owl:sameAs->?o]
eq-sym	?x[owl:sameAs->?y]	?y[owl:sameAs->?x]
prp-ap-l†		rdfs:label[rdf:type-> owl:AnnotationProperty]
prp-ap-c†		rdfs:comment[rdf:type-> owl:AnnotationProperty]
prp-ap-sa†		rdfs:seeAlso[rdf:type-> owl:AnnotationProperty]
prp-ap-idb†		rdfs:isDefinedBy[rdf:type-> owl:AnnotationProperty]
prp-ap-d†		owl:deprecated[rdf:type-> owl:AnnotationProperty]
prp-ap-pv†		owl:priorVersion[rdf:type-> owl:AnnotationProperty]
prp-ap-bcw†		owl:backwardCompatibleWith[rdf:type-> owl:AnnotationProperty]
prp-ap-iw†		owl:incompatibleWith[rdf:type-> owl:AnnotationProperty]
prp-dom*	p[rdfs:domain->?c] ?x[?p->?y] ?c $\notin MC$	?x[rdf:type->?c]
prp-rng*	?p[rdfs:range->?c] ?x[?p->?y] ?c $\notin MC$	?y[rdf:type->?c]
prp-irp	?p[rdf:type->owl:IrreflexiveProperty] ?x[?p->?x]	rif:error()
prp-symp1*	?p[rdf:type->owl:SymmetricProperty] ?x[?p->?y] ?p $\notin MPT$	?y[?p->?x]

prp-symp2*	?p[rdf:type->owl:SymmetricProperty] ?x[rdf:type->?y] ?x $\notin MC$	?y[rdf:type->?x]
prp-trp*	?p[rdf:type->owl:TransitiveProperty] ?x[?p->?y] ?y[?p->?z] ?p $\in$ MP	?x[?p->?z]
prp-spo11*	?p1[rdfs:subPropertyOf->?p2] ?x[?p1->?y] ?p2 $\notin MPT$	?x[?p2->?y]
prp-spo12*	?p1[rdfs:subPropertyOf->rdf:type] ?x[?p1->?y] ?y $\notin MC$	?x[rdf:type->?y]
prp-eqp11*	?p1[owl:equivalentProperty->?p2] ?x[?p1->?y] ?p2 $\notin MPT$	?x[?p2->?y]
prp-eqp12*	?p1[owl:equivalentProperty->rdf:type] ?x[?p1->?y] ?y $\notin MC$	?x[rdf:type->?y]
prp-eqp21*	?p1[owl:equivalentProperty->?p2] ?x[?p2->?y] ?p1 $\notin MPT$	?x[?p1->?y]
prp-eqp22*	rdf:type[owl:equivalentProperty->?p2] ?x[?p2->?y] ?y $\notin MC$	?x[rdf:type->?y]
prp-inv11*	?p1[owl:inverseOf->?p2] ?x[?p1->?y] ?p2 $\notin MPT$	?y[?p2->?x]
prp-inv12*	?p1[owl:inverseOf->rdf:type] ?x[?p1->?y] ?x $\notin MC$	?y[rdf:type->?x]
prp-inv21*	?p1[owl:inverseOf->?p2] ?x[?p2->?y] ?p1 $\notin MPT$	?y[?p1->?x]
prp-inv22*	rdf:type[owl:inverseOf->?p2] ?x[?p2->?y] ?x $\notin MC$	?y[rdf:type->?x]
cls-thing†		owl:Thing[rdf:type->owl:Class]
cls-nothing1†		owl:Nothing[rdf:type->owl:Class]
cls-nothing2	?x[rdf:type->owl:Nothing]	rif:error()
cls-svf2*	?x[owl:someValuesFrom->owl:Thing] ?x[owl:onProperty->?p] ?u[?p->?v] ?x $\notin MC$	?u[rdf:type->?x]
cls-hv11*	?x[owl:hasValue->?y] ?x[owl:onProperty->?p]	?u[?p->?y]

	?u[rdf:type->?x] ?p $\notin MPT$	
cls-hv12*	?x[owl:hasValue->?y] ?x[owl:onProperty->rdf:type] ?u[rdf:type->?x] ?y $\notin MC$	?u[rdf:type->?y]
cls-hv2*	?x[owl:hasValue->?y] ?x[owl:onProperty->?p] ?u[?p->y] ?x $\notin MC$	?u[rdf:type->?y]
cax-sco*	?c1[rdfs:subClassOf->?c2] ?x[rdf:type->?c1] ?c2 $\notin MC$	?x[rdf:type->?c2]
cax-eqc1*	?c1[owl:equivalentClass->?c2] ?x[rdf:type->?c1] ?c2 $\notin MC$	?x[rdf:type->?c2]
cax-eqc2*	?c1[owl:equivalentClass->?c2] ?x[rdf:type->?c2] ?c1 $\notin MC$	?x[rdf:type->?c1]
eq-diff2a	?x[rdf:type->owl:AllDifferent] ?x[owl:distinctMembers->?y]	_checkDifferent(?x ?y)
eq-diff3a	?x[rdf:type->owl:AllDifferent] ?x[owl:members->?y]	_checkDifferent(?x ?y)
eq-diff23b	_checkDifferent(?x ?z) ?z[rdf:rest->?y] Not(?y = rdf:nil)	_checkDifferent(?x ?y)
eq-diff23c	_checkDifferent(?x ?y1) _checkDifferent(?x ?y2) Not(?y1 = ?y2) ?y1[rdf:first->?z1] ?y2[rdf:first->?z2] ?z1[owl:sameAs->?z2]	rif:error()
cls-uni*	_checkUnionOf(?c ?l) ?l[rdf:first->?ci] ?y[rdf:type->?ci] ?c $\notin MC$	?y[rdf:type->?c]
cls-oo-a	?c[owl:oneOf->?l]	_checkOneOf(?c ?l)
cls-oo-b	_checkOneOf(?c ?r) ?r[rdf:rest-?l] Not(?l = rdf:nil)	_checkOneOf(?c ?l)
cls-oo-c1*	_checkOneOf(?c ?l) ?l[rdf:first->?yi] ?c $\notin MC$	?yi[rdf:type->?c]
cls-oo-c2*	_checkOneOf(?c ?l) ?l[rdf:first->?yi] ?c $\in MC$	?yi[rdf:type->?c]
cls-int2*	_markAllTypes(?c ?l)	?y[rdf:type->?ci]

	?l[rdf:first->?ci] ?y[rdf:type->?c] ?ci $\notin MC$	

B.3 Verbatim Rulesets Used in Evaluation

For the purposes of reproducibility, this section simply includes the rulesets as they were provided to the inference engine. They are specified in a modified RIF-Core syntax which is likely intuitively understandable to those familiar with RIF-Core. Note that the # symbol is used at the beginning of lines to indicate comments, some of which are interpreted by the rule compiler (e.g., `#DEFINE` and `#PRAGMA`). `#PRAGMA` is used to specify patterns for replication or arbitrary placement. Note that unlike the definition of Pattern used in this thesis which utilizes negated equality formulas as restrictions, the rules below instead specify restrictions by negating a built-in formula with predicate `pred:list-contains`. This detail is obscured, however, by using symbols `IN` and `NOTIN`. `#DEFINE` has similar behavior to `#define` in C++. Note that rule labels given in `(* *)` may not necessarily be consistent with the names used throughout the rest of this thesis.

B.3.1 Par-CoreRDFS

```
Prefix(rdf  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(rdfs <http://www.w3.org/2000/01/rdf-schema#>)
Prefix(owl  <http://www.w3.org/2002/07/owl#>)
Prefix(xsd  <http://www.w3.org/2001/XMLSchema#>)
Prefix(rif  <http://www.w3.org/2007/rif#>)
Prefix(func <http://www.w3.org/2007/rif-builtin-function#>)
Prefix(pred <http://www.w3.org/2007/rif-builtin-predicate#>)
Prefix(dc   <http://purl.org/dc/terms/>)

#DEFINE IN External(pred:list-contains(
#DEFINE /IN ))
#DEFINE NOTIN Not({IN}
#DEFINE /NOTIN {/IN})
#DEFINE $MP List(rdfs:domain rdfs:range rdfs:subPropertyOf rdfs:subClassOf)

#PRAGMA REPLICATE And(?p[rdfs:domain -> ?c])
#PRAGMA REPLICATE And(?p[rdfs:range -> ?c])
#PRAGMA REPLICATE And(?p1[rdfs:subPropertyOf -> ?p2])
#PRAGMA REPLICATE And(?c1[rdfs:subClassOf -> ?c2])
#PRAGMA ARBITRARY And(?x[?p2 -> ?y] {NOTIN}{$MP} ?p2{/NOTIN})
```

```
(* <#scm-spo> *)
Forall ?p3 ?p2 ?p1 (
   ?p1[rdfs:subPropertyOf->?p3] :- And(
       ?p1[rdfs:subPropertyOf->?p2]
       ?p2[rdfs:subPropertyOf->?p3]  ))

(* <#scm-sco> *)
Forall ?c1 ?c2 ?c3 (
   ?c1[rdfs:subClassOf->?c3] :- And(
       ?c1[rdfs:subClassOf->?c2]
       ?c2[rdfs:subClassOf->?c3]  ))

(* <#prp-spo1> *)
Forall ?x ?y ?p2 ?p1 (
   ?x[?p2->?y] :- And(
       ?p1[rdfs:subPropertyOf->?p2]
       {NOTIN}{$MP} ?p2{/NOTIN}
       ?x[?p1->?y]  ))

(* <#prp-dom> *)
Forall ?p ?c ?x ?y (
   ?x[rdf:type->?c] :- And(
       ?p[rdfs:domain->?c]
       ?x[?p->?y]  ))

(* <#prp-rng> *)
Forall ?p ?c ?x ?y (
   ?y[rdf:type->?c] :- And(
       ?p[rdfs:range->?c]
       ?x[?p->?y]  ))

(* <#cax-sco> *)
Forall ?x ?c1 ?c2 (
   ?x[rdf:type->?c2] :- And(
       ?c1[rdfs:subClassOf->?c2]
       ?x[rdf:type->?c1]  ))

#EOF
```

B.3.2 Par-MemOWL2

```
Prefix(rdf  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
 Prefix(rdfs <http://www.w3.org/2000/01/rdf-schema#>)
 Prefix(owl  <http://www.w3.org/2002/07/owl#>)
 Prefix(xsd  <http://www.w3.org/2001/XMLSchema#>)
 Prefix(rif  <http://www.w3.org/2007/rif#>)
 Prefix(func <http://www.w3.org/2007/rif-builtin-function#>)
```

```
 Prefix(pred <http://www.w3.org/2007/rif-builtin-predicate#>)
 Prefix(dc   <http://purl.org/dc/terms/>)

#DEFINE IN External(pred:list-contains(
#DEFINE /IN ))
#DEFINE NOTIN Not({IN}
#DEFINE /NOTIN {/IN})

#DEFINE $MC List(owl:FunctionalProperty owl:InverseFunctionalProperty
 owl:IrreflexiveProperty owl:SymmetricProperty owl:AsymmetricProperty
 owl:TransitiveProperty owl:Class owl:ObjectProperty owl:DatatypeProperty
 owl:AllDifferent owl:AllDisjointProperties owl:AllDisjointClasses)
#DEFINE $MP List(rdfs:domain rdfs:range rdfs:subPropertyOf owl:equivalentProperty
 owl:propertyDisjointWith owl:inverseOf owl:someValuesFrom owl:onProperty
 owl:allValuesFrom owl:hasValue owl:onClass rdfs:subClassOf owl:equivalentClass
 owl:disjointWith owl:complementOf owl:distinctMembers owl:members
 owl:propertyChainAxiom owl:intersectionOf owl:hasKey owl:unionOf owl:oneOf
 rdf:first rdf:rest owl:maxCardinality owl:maxQualifiedCardinality)
#DEFINE $MPT List(rdfs:domain rdfs:range rdfs:subPropertyOf owl:equivalentProperty
 owl:propertyDisjointWith owl:inverseOf owl:someValuesFrom owl:onProperty
 owl:allValuesFrom owl:hasValue owl:onClass rdfs:subClassOf owl:equivalentClass
 owl:disjointWith owl:complementOf owl:distinctMembers owl:members
 owl:propertyChainAxiom owl:intersectionOf owl:hasKey owl:unionOf owl:oneOf
 rdf:first rdf:rest owl:maxCardinality owl:maxQualifiedCardinality rdf:type)

# REPLICATE ONTOLOGY
#PRAGMA REPLICATE And(?s[?p->?o] {IN}{$MP} ?p{/IN})
#PRAGMA REPLICATE And(?s[rdf:type->?o] {IN}{$MC} ?o{/IN})
#PRAGMA REPLICATE And(_markAllTypes(?a ?b))
#PRAGMA REPLICATE And(_allTypes(?a ?b ?c))
#PRAGMA REPLICATE And(_checkUnionOf(?a ?b))
#PRAGMA REPLICATE And(_checkDifferent(?a ?b))
#PRAGMA REPLICATE And(_checkDisjointProperties(?a ?b))
#PRAGMA REPLICATE And(_checkDisjointClasses(?a ?b))
#PRAGMA REPLICATE And(_markCheckChain(?a ?b))
#PRAGMA REPLICATE And(_checkChain(?a ?b ?c ?d))
#PRAGMA REPLICATE And(_markSameKey(?a ?b))
#PRAGMA REPLICATE And(_sameKey(?a ?b ?c ?d))

# REPLICATE BUILTINS (pred:list-contains special, don't worry about it)

# REPLICATE SELECTIVE PATTERNS
#PRAGMA REPLICATE And(?a = ?b)

#PRAGMA ARBITRARY And(?s[?p->?o] {NOTIN}{$MPT} ?p{/NOTIN})
#PRAGMA ARBITRARY And(?s[rdf:type->?o] {NOTIN}{$MC} ?o{/NOTIN})
```

```
(* <#scm-int> *)
 Forall ?c ?l (
     _markAllTypes(?c ?l) :- ?c[owl:intersectionOf->?l] )

 Forall ?c ?l ?r (
     _markAllTypes(?c ?l) :- And (
         _markAllTypes(?c ?r)
         ?r[rdf:rest->?l]
         Not(?l = rdf:nil) ))

 Forall ?c ?ci ?l (
   ?c[rdfs:subClassOf->?ci] :- And (
         _markAllTypes(?c ?l)
         ?l[rdf:first->?ci] ))

(* <#scm-uni> *)
 Forall ?c ?l (
     _checkUnionOf(?c ?l) :- ?c[owl:unionOf->?l] )

 Forall ?c ?l ?r (
     _checkUnionOf(?c ?l) :- And(
         _checkUnionOf(?c ?r)
         ?r[rdf:rest->?l]
         Not(?l = rdf:nil) ))

 Forall ?c ?ci ?l (
   ?ci[rdfs:subClassOf->?c] :- And (
         _checkUnionOf(?c ?l)
         ?l[rdf:first->?ci] ))

#UNINTERESTING
#(* <#scm-cls> *)
#Forall ?c (
#   ?c[rdfs:subClassOf->?c] :- ?c[rdf:type->owl:Class])

#UNINTERESTING
#(* <#scm-cls1> *)
#Forall ?c (
#   ?c[owl:equivalentClass->?c] :- ?c[rdf:type->owl:Class])

#NOT WORTH THE MEMORY
#(* <#scm-cls2> *)
#Forall ?c (
#   ?c[rdfs:subClassOf->owl:Thing] :- ?c[rdf:type->owl:Class])

#NOT WORTH THE MEMORY
#(* <#scm-cls3> *)
```

```
#Forall ?c (
#   owl:Nothing[rdfs:subClassOf->?c] :- ?c[rdf:type->owl:Class])

(* <#scm-sco> *)
Forall ?c1 ?c2 ?c3 (
   ?c1[rdfs:subClassOf->?c3] :- And(
       ?c1[rdfs:subClassOf->?c2]
       ?c2[rdfs:subClassOf->?c3]  ))

(* <#scm-eqc1> *)
Forall ?c1 ?c2 (
   ?c1[rdfs:subClassOf->?c2] :- ?c1[owl:equivalentClass->?c2])

(* <#scm-eqc11> *)
Forall ?c1 ?c2 (
   ?c2[rdfs:subClassOf->?c1] :- ?c1[owl:equivalentClass->?c2])

(* <#scm-eqc2> *)
Forall ?c1 ?c2 (
   ?c1[owl:equivalentClass->?c2] :- And(
       ?c1[rdfs:subClassOf->?c2]
       ?c2[rdfs:subClassOf->?c1]  ))

#UNINTERESTING
#(* <#scm-op> *)
#Forall ?p (
#   ?p[rdfs:subPropertyOf->?p] :- ?p[rdf:type->owl:ObjectProperty])

#UNINTERESTING
#(* <#scm-op1> *)
#Forall ?p (
#   ?p[owl:equivalentProperty->?p] :- ?p[rdf:type->owl:ObjectProperty])

#UNINTERESTING
#(* <#scm-dp> *)
#Forall ?p (
#   ?p[rdfs:subPropertyOf->?p] :- ?p[rdf:type->owl:DatatypeProperty])

#UNINTERESTING
#(* <#scm-dp1> *)
#Forall ?p (
#   ?p[owl:equivalentProperty->?p] :- ?p[rdf:type->owl:DatatypeProperty])

(* <#scm-spo> *)
Forall ?p3 ?p2 ?p1 (
   ?p1[rdfs:subPropertyOf->?p3] :- And(
       ?p1[rdfs:subPropertyOf->?p2]
```

```
       ?p2[rdfs:subPropertyOf->?p3]  ))

(* <#scm-eqp1> *)
Forall ?p2 ?p1 (
   ?p1[rdfs:subPropertyOf->?p2] :- ?p1[owl:equivalentProperty->?p2])

(* <#scm-eqp11> *)
Forall ?p2 ?p1 (
   ?p2[rdfs:subPropertyOf->?p1] :- ?p1[owl:equivalentProperty->?p2])

(* <#scm-eqp2> *)
Forall ?p2 ?p1 (
   ?p1[owl:equivalentProperty->?p2] :- And(
       ?p1[rdfs:subPropertyOf->?p2]
       ?p2[rdfs:subPropertyOf->?p1]  ))

(* <#scm-dom1> *)
Forall ?p ?c1 ?c2 (
   ?p[rdfs:domain->?c2] :- And(
       ?p[rdfs:domain->?c1]
       ?c1[rdfs:subClassOf->?c2]  ))

(* <#scm-dom2> *)
Forall ?c ?p2 ?p1 (
   ?p1[rdfs:domain->?c] :- And(
       ?p2[rdfs:domain->?c]
       ?p1[rdfs:subPropertyOf->?p2]  ))

(* <#scm-rng1> *)
Forall ?p ?c1 ?c2 (
   ?p[rdfs:range->?c2] :- And(
       ?p[rdfs:range->?c1]
       ?c1[rdfs:subClassOf->?c2]  ))

(* <#scm-rng2> *)
Forall ?c ?p2 ?p1 (
   ?p1[rdfs:range->?c] :- And(
       ?p2[rdfs:range->?c]
       ?p1[rdfs:subPropertyOf->?p2]  ))

(* <#scm-hv> *)
Forall ?c1 ?c2 ?i ?p2 ?p1 (
   ?c1[rdfs:subClassOf->?c2] :- And(
       ?c1[owl:hasValue->?i]
       ?c1[owl:onProperty->?p1]
       ?c2[owl:hasValue->?i]
       ?c2[owl:onProperty->?p2]
```

```
       ?p1[rdfs:subPropertyOf->?p2]  ))

(* <#scm-svf1> *)
Forall ?p ?y2 ?c1 ?c2 ?y1 (
   ?c1[rdfs:subClassOf->?c2] :- And(
       ?c1[owl:someValuesFrom->?y1]
       ?c1[owl:onProperty->?p]
       ?c2[owl:someValuesFrom->?y2]
       ?c2[owl:onProperty->?p]
       ?y1[rdfs:subClassOf->?y2]  ))

(* <#scm-svf2> *)
Forall ?c1 ?c2 ?y ?p2 ?p1 (
   ?c1[rdfs:subClassOf->?c2] :- And(
       ?c1[owl:someValuesFrom->?y]
       ?c1[owl:onProperty->?p1]
       ?c2[owl:someValuesFrom->?y]
       ?c2[owl:onProperty->?p2]
       ?p1[rdfs:subPropertyOf->?p2]  ))

(* <#scm-avf1> *)
Forall ?p ?y2 ?c1 ?c2 ?y1 (
   ?c1[rdfs:subClassOf->?c2] :- And(
       ?c1[owl:allValuesFrom->?y1]
       ?c1[owl:onProperty->?p]
       ?c2[owl:allValuesFrom->?y2]
       ?c2[owl:onProperty->?p]
       ?y1[rdfs:subClassOf->?y2]  ))

(* <#scm-avf2> *)
Forall ?c1 ?c2 ?y ?p2 ?p1 (
   ?c2[rdfs:subClassOf->?c1] :- And(
       ?c1[owl:allValuesFrom->?y]
       ?c1[owl:onProperty->?p1]
       ?c2[owl:allValuesFrom->?y]
       ?c2[owl:onProperty->?p2]
       ?p1[rdfs:subPropertyOf->?p2]  ))

#NOT WORTH THE MEMORY
#(* <#eq-ref> *)
#Forall ?p ?o ?s (
#   ?s[owl:sameAs->?s] :- ?s[?p->?o])

#NOT WORTH THE MEMORY
#(* <#eq-ref1> *)
#Forall ?p ?o ?s (
#   ?p[owl:sameAs->?p] :- ?s[?p->?o])
```

```
#NOT WORTH THE MEMORY
#(* <#eq-ref2> *)
#Forall ?p ?o ?s (
#   ?o[owl:sameAs->?o] :- ?s[?p->?o])

(* <#eq-sym> *)
Forall ?x ?y (
   ?y[owl:sameAs->?x] :- ?x[owl:sameAs->?y])

#ELIMINATED
#(* <#eq-trans> *)
#Forall ?x ?z ?y (
#   ?x[owl:sameAs->?z] :- And(
#       ?x[owl:sameAs->?y]
#       ?y[owl:sameAs->?z]  ))

#SPLIT
#ELIMINATED
#(* <#eq-rep-s> *)
#Forall ?p ?o ?s ?s2 (
#   ?s2[?p->?o] :- And(
#       ?s[owl:sameAs->?s2]
#       ?s[?p->?o]
#             {NOTIN}{$MPT} ?p{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?s2 (
#   ?s2[?p->?o] :- And(
#       ?s[owl:sameAs->?s2]
#       ?s[?p->?o]
#             {IN}{$MP} ?p{/IN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?s2 (
#   ?s2[rdf:type->?o] :- And(
#       ?s[owl:sameAs->?s2]
#       ?s[rdf:type->?o]
#             {NOTIN}{$MC} ?o{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?s2 (
#   ?s2[rdf:type->?o] :- And(
#       ?s[owl:sameAs->?s2]
#       ?s[rdf:type->?o]
#             {IN}{$MC} ?o{/IN}  ))
```

```
#SPLIT
#ELIMINATED
#(* <#eq-rep-p> *)
#Forall ?p ?o ?s ?p2 (
#   ?s[?p2->?o] :- And(
#       ?p[owl:sameAs->?p2]
#       ?s[?p->?o]
#             {NOTIN}{$MPT} ?p2{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?p2 (
#   ?s[?p2->?o] :- And(
#       ?p[owl:sameAs->?p2]
#       ?s[?p->?o]
#             {IN}{$MP} ?p2{/IN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?p2 (
#   ?s[rdf:type->?o] :- And(
#       ?p[owl:sameAs->rdf:type]
#       ?s[?p->?o]
#             {NOTIN}{$MC} ?o{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?p2 (
#   ?s[rdf:type->?o] :- And(
#       ?p[owl:sameAs->rdf:type]
#       ?s[?p->?o]
#             {IN}{$MC} ?o{/IN}  ))

#SPLIT
#ELIMINATED
#(* <#eq-rep-o> *)
#Forall ?p ?o ?s ?o2 (
#   ?s[?p->?o2] :- And(
#       ?o[owl:sameAs->?o2]
#       ?s[?p->?o]
#             {NOTIN}{$MPT} ?p{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?o2 (
#   ?s[?p->?o2] :- And(
#       ?o[owl:sameAs->?o2]
#       ?s[?p->?o]
#             {IN}{$MP} ?p{/IN}  ))

#ELIMINATED
```

```
#Forall ?p ?o ?s ?o2 (
#   ?s[rdf:type->?o2] :- And(
#       ?o[owl:sameAs->?o2]
#       ?s[rdf:type->?o]
#             {NOTIN}{$MC} ?o2{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?o ?s ?o2 (
#   ?s[rdf:type->?o2] :- And(
#       ?o[owl:sameAs->?o2]
#       ?s[rdf:type->?o]
#             {IN}{$MC} ?o2{/IN}  ))

#ELIMINATED
#(* <#eq-diff1> *)
#Forall ?x ?y (
#   rif:error() :- And(
#       ?x[owl:sameAs->?y]
#       ?x[owl:differentFrom->?y]  ))

#UNINTERESTING
#(* <#prp-ap-label> *)
#    rdfs:label[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-comment> *)
#   rdfs:comment[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-seeAlso> *)
#   rdfs:seeAlso[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-isDefinedBy> *)
#   rdfs:isDefinedBy[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-deprecated> *)
#   owl:deprecated[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-priorVersion> *)
#   owl:priorVersion[rdf:type->owl:AnnotationProperty]

#UNINTERESTING
#(* <#prp-ap-backwardCompatibleWith> *)
#   owl:backwardCompatibleWith[rdf:type->owl:AnnotationProperty]
```

```
#UNINTERESTING
#(* <#prp-ap-incompatibleWith> *)
#   owl:incompatibleWith[rdf:type->owl:AnnotationProperty]

#SPLIT
(* <#prp-dom> *)
Forall ?p ?c ?x ?y (
   ?x[rdf:type->?c] :- And(
       ?p[rdfs:domain->?c]
       ?x[?p->?y]
             {NOTIN}{$MC} ?c{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?c ?x ?y (
#   ?x[rdf:type->?c] :- And(
#       ?p[rdfs:domain->?c]
#       ?x[?p->?y]
#             {IN}{$MC} ?c{/IN}  ))

#SPLIT
(* <#prp-rng> *)
Forall ?p ?c ?x ?y (
   ?y[rdf:type->?c] :- And(
       ?p[rdfs:range->?c]
       ?x[?p->?y]
             {NOTIN}{$MC} ?c{/NOTIN}  ))

#ELIMINATED
#(* <#prp-rng> *)
#Forall ?p ?c ?x ?y (
#   ?y[rdf:type->?c] :- And(
#       ?p[rdfs:range->?c]
#       ?x[?p->?y]
#             {IN}{$MC} ?c{/IN}  ))

#ELIMINATED
#(* <#prp-fp> *)
#Forall ?p ?y2 ?x ?y1 (
#   ?y1[owl:sameAs->?y2] :- And(
#       ?p[rdf:type->owl:FunctionalProperty]
#       ?x[?p->?y1]
#       ?x[?p->?y2]  ))

#ELIMINATED
#(* <#prp-ifp> *)
#Forall ?p ?x1 ?x2 ?y (
```

```
#   ?x1[owl:sameAs->?x2] :- And(
#       ?p[rdf:type->owl:InverseFunctionalProperty]
#       ?x1[?p->?y]
#       ?x2[?p->?y]  ))

(* <#prp-irp> *)
Forall ?p ?x (
   rif:error() :- And(
       ?p[rdf:type->owl:IrreflexiveProperty]
       ?x[?p->?x]  ))

#SPLIT
(* <#prp-symp> *)
Forall ?p ?x ?y (
   ?y[?p->?x] :- And(
       ?p[rdf:type->owl:SymmetricProperty]
       ?x[?p->?y]
             {NOTIN}{$MPT} ?p{/NOTIN}  ))

Forall ?p ?x ?y (
   ?y[?p->?x] :- And(
       ?p[rdf:type->owl:SymmetricProperty]
             {IN}{$MP} ?p{/IN}
       ?x[?p->?y]  ))

Forall ?p ?x ?y (
   ?y[rdf:type->?x] :- And(
       rdf:type[rdf:type->owl:SymmetricProperty]
       ?x[rdf:type->?y]
             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?x ?y (
#   ?y[rdf:type->?x] :- And(
#       rdf:type[rdf:type->owl:SymmetricProperty]
#       ?x[rdf:type->?y]
#             {IN}{$MC} ?x{/IN}  ))

#ELIMINATED
#(* <#prp-asyp> *)
#Forall ?p ?x ?y (
#   rif:error() :- And(
#       ?p[rdf:type->owl:AsymmetricProperty]
#       ?x[?p->?y]
#       ?y[?p->?x]  ))

#SPLIT
```

```
#ELIMINATED
#(* <#prp-trp> *)
#Forall ?p ?x ?z ?y (
#   ?x[?p->?z] :- And(
#       ?p[rdf:type->owl:TransitiveProperty]
#       ?x[?p->?y]
#       ?y[?p->?z]
#             {NOTIN}{$MPT} ?p{/NOTIN}  ))

Forall ?p ?x ?z ?y (
   ?x[?p->?z] :- And(
       ?p[rdf:type->owl:TransitiveProperty]
            {IN}{$MP} ?p{/IN}
       ?x[?p->?y]
       ?y[?p->?z]  ))

#ELIMINATED
#Forall ?p ?x ?z ?y (
#   ?x[rdf:type->?z] :- And(
#       rdf:type[rdf:type->owl:TransitiveProperty]
#       ?x[rdf:type->?y]
#       ?y[rdf:type->?z]
#             {NOTIN}{$MC} ?z{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?x ?z ?y (
#   ?x[rdf:type->?z] :- And(
#       rdf:type[rdf:type->owl:TransitiveProperty]
#       ?x[rdf:type->?y]
#       ?y[rdf:type->?z]
#             {IN}{$MC} ?z{/IN}  ))

#SPLIT
(* <#prp-spo1> *)
Forall ?x ?y ?p2 ?p1 (
   ?x[?p2->?y] :- And(
       ?p1[rdfs:subPropertyOf->?p2]
       ?x[?p1->?y]
            {NOTIN}{$MPT} ?p2{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[?p2->?y] :- And(
#       ?p1[rdfs:subPropertyOf->?p2]
#       ?x[?p1->?y]
#             {IN}{$MP} ?p2{/IN}  ))
```

```
Forall ?x ?y ?p2 ?p1 (
   ?x[rdf:type->?y] :- And(
       ?p1[rdfs:subPropertyOf->rdf:type]
       ?x[?p1->?y]
             {NOTIN}{$MC} ?y{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[rdf:type->?y] :- And(
#       ?p1[rdfs:subPropertyOf->rdf:type]
#       ?x[?p1->?y]
#             {IN}{$MC} ?y{/IN}  ))

#SPLIT
(* <#prp-eqp1> *)
Forall ?x ?y ?p2 ?p1 (
   ?x[?p2->?y] :- And(
       ?p1[owl:equivalentProperty->?p2]
       ?x[?p1->?y]
             {NOTIN}{$MPT} ?p2{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[?p2->?y] :- And(
#       ?p1[owl:equivalentProperty->?p2]
#       ?x[?p1->?y]
#             {IN}{$MP} ?p2{/IN}  ))

Forall ?x ?y ?p2 ?p1 (
   ?x[rdf:type->?y] :- And(
       ?p1[owl:equivalentProperty->rdf:type]
       ?x[?p1->?y]
             {NOTIN}{$MC} ?y{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[rdf:type->?y] :- And(
#       ?p1[owl:equivalentProperty->rdf:type]
#       ?x[?p1->?y]
#             {IN}{$MC} ?y{/IN}  ))

#SPLIT
(* <#prp-eqp2> *)
Forall ?x ?y ?p2 ?p1 (
   ?x[?p1->?y] :- And(
       ?p1[owl:equivalentProperty->?p2]
       ?x[?p2->?y]
```

```
            {NOTIN}{$MPT} ?p1{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[?p1->?y] :- And(
#       ?p1[owl:equivalentProperty->?p2]
#       ?x[?p2->?y]
#             {IN}{$MP} ?p1{/IN}  ))

Forall ?x ?y ?p2 ?p1 (
   ?x[rdf:type->?y] :- And(
       rdf:type[owl:equivalentProperty->?p2]
       ?x[?p2->?y]
             {NOTIN}{$MC} ?y{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?x[rdf:type->?y] :- And(
#       rdf:type[owl:equivalentProperty->?p2]
#       ?x[?p2->?y]
#             {IN}{$MC} ?y{/IN}  ))

#ELIMINATED
#(* <#prp-pdw> *)
#Forall ?x ?y ?p2 ?p1 (
#   rif:error() :- And(
#       ?p1[owl:propertyDisjointWith->?p2]
#       ?x[?p1->?y]
#       ?x[?p2->?y]  ))

#SPLIT
(* <#prp-inv1> *)
Forall ?x ?y ?p2 ?p1 (
   ?y[?p2->?x] :- And(
       ?p1[owl:inverseOf->?p2]
       ?x[?p1->?y]
             {NOTIN}{$MPT} ?p2{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?y[?p2->?x] :- And(
#       ?p1[owl:inverseOf->?p2]
#       ?x[?p1->?y]
#             {IN}{$MP} ?p2{/IN}  ))

Forall ?x ?y ?p2 ?p1 (
   ?y[rdf:type->?x] :- And(
```

```
       ?p1[owl:inverseOf->rdf:type]
       ?x[?p1->?y]
             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?y[rdf:type->?x] :- And(
#       ?p1[owl:inverseOf->rdf:type]
#       ?x[?p1->?y]
#             {IN}{$MC} ?x{/IN}  ))

#SPLIT
(* <#prp-inv2> *)
Forall ?x ?y ?p2 ?p1 (
   ?y[?p1->?x] :- And(
       ?p1[owl:inverseOf->?p2]
       ?x[?p2->?y]
             {NOTIN}{$MPT} ?p1{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?y[?p1->?x] :- And(
#       ?p1[owl:inverseOf->?p2]
#       ?x[?p2->?y]
#             {IN}{$MP} ?p1{/IN}  ))

Forall ?x ?y ?p2 ?p1 (
   ?y[rdf:type->?x] :- And(
       rdf:type[owl:inverseOf->?p2]
       ?x[?p2->?y]
             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?y ?p2 ?p1 (
#   ?y[rdf:type->?x] :- And(
#       rdf:type[owl:inverseOf->?p2]
#       ?x[?p2->?y]
#             {IN}{$MC} ?x{/IN}  ))

#UNINTERESTING
#(* <#cls-thing> *)
#   owl:Thing[rdf:type->owl:Class]

#UNINTERESTING
#(* <#cls-nothing1> *)
#    owl:Nothing[rdf:type->owl:Class]
```

```
(* <#cls-nothing2> *)
Forall ?x (
   rif:error() :- ?x[rdf:type->owl:Nothing])

#SPLIT
#ELIMINATED
#(* <#cls-svf1> *)
#Forall ?p ?v ?u ?x ?y (
#   ?u[rdf:type->?x] :- And(
#       ?x[owl:someValuesFrom->?y]
#       ?x[owl:onProperty->?p]
#       ?u[?p->?v]
#       ?v[rdf:type->?y]
#             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?v ?u ?x ?y (
#   ?u[rdf:type->?x] :- And(
#       ?x[owl:someValuesFrom->?y]
#       ?x[owl:onProperty->?p]
#       ?u[?p->?v]
#       ?v[rdf:type->?y]
#             {IN}{$MC} ?x{/IN}  ))

#SPLIT
(* <#cls-svf2> *)
Forall ?p ?v ?u ?x (
   ?u[rdf:type->?x] :- And(
       ?x[owl:someValuesFrom->owl:Thing]
       ?x[owl:onProperty->?p]
       ?u[?p->?v]
             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?v ?u ?x (
#   ?u[rdf:type->?x] :- And(
#       ?x[owl:someValuesFrom->owl:Thing]
#       ?x[owl:onProperty->?p]
#       ?u[?p->?v]
#             {IN}{$MC} ?x{/IN}  ))

#SPLIT
#ELIMINATED
#(* <#cls-avf> *)
#Forall ?p ?v ?u ?x ?y (
#   ?v[rdf:type->?y] :- And(
#       ?x[owl:allValuesFrom->?y]
```

```
#       ?x[owl:onProperty->?p]
#       ?u[rdf:type->?x]
#       ?u[?p->?v]
#             {NOTIN}{$MC} ?y{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?v ?u ?x ?y (
#   ?v[rdf:type->?y] :- And(
#       ?x[owl:allValuesFrom->?y]
#       ?x[owl:onProperty->?p]
#       ?u[rdf:type->?x]
#       ?u[?p->?v]
#             {IN}{$MC} ?y{/IN}  ))

#SPLIT
(* <#cls-hv1> *)
Forall ?p ?u ?x ?y (
   ?u[?p->?y] :- And(
       ?x[owl:hasValue->?y]
       ?x[owl:onProperty->?p]
       ?u[rdf:type->?x]
             {NOTIN}{$MPT} ?p{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?u ?x ?y (
#   ?u[?p->?y] :- And(
#       ?x[owl:hasValue->?y]
#       ?x[owl:onProperty->?p]
#       ?u[rdf:type->?x]
#             {IN}{$MP} ?p{/IN}  ))

Forall ?p ?u ?x ?y (
   ?u[rdf:type->?y] :- And(
       ?x[owl:hasValue->?y]
       ?x[owl:onProperty->rdf:type]
       ?u[rdf:type->?x]
             {NOTIN}{$MC} ?y{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?u ?x ?y (
#   ?u[rdf:type->?y] :- And(
#       ?x[owl:hasValue->?y]
#       ?x[owl:onProperty->rdf:type]
#       ?u[rdf:type->?x]
#             {IN}{$MC} ?y{/IN}  ))

#SPLIT
```

```
(* <#cls-hv2> *)
Forall ?p ?u ?x ?y (
   ?u[rdf:type->?x] :- And(
       ?x[owl:hasValue->?y]
       ?x[owl:onProperty->?p]
       ?u[?p->?y]
             {NOTIN}{$MC} ?x{/NOTIN}  ))

#ELIMINATED
#Forall ?p ?u ?x ?y (
#   ?u[rdf:type->?x] :- And(
#       ?x[owl:hasValue->?y]
#       ?x[owl:onProperty->?p]
#       ?u[?p->?y]
#             {IN}{$MC} ?x{/IN}  ))

#ELIMINATED
#(* <#cls-maxc1> *)
#Forall ?p ?u ?x ?y (
#   rif:error() :- And(
#       ?x[owl:maxCardinality->0]
#       ?x[owl:onProperty->?p]
#       ?u[rdf:type->?x]
#       ?u[?p->?y]  ))

#ELIMINATED
#(* <#cls-maxc2> *)
#Forall ?p ?y2 ?u ?x ?y1 (
#   ?y1[owl:sameAs->?y2] :- And(
#       ?x[owl:maxCardinality->1]
#       ?x[owl:onProperty->?p]
#       ?u[rdf:type->?x]
#       ?u[?p->?y1]
#       ?u[?p->?y2]  ))

#ELIMINATED
#(* <#cls-maxqc1> *)
#Forall ?p ?c ?u ?x ?y (
#   rif:error() :- And(
#       ?x[owl:maxQualifiedCardinality->0]
#       ?x[owl:onProperty->?p]
#       ?x[owl:onClass->?c]
#       ?u[rdf:type->?x]
#       ?u[?p->?y]
#       ?y[rdf:type->?c]  ))

#ELIMINATED
```

```
#(* <#cls-maxqc2> *)
#Forall ?p ?u ?x ?y (
#   rif:error() :- And(
#       ?x[owl:maxQualifiedCardinality->0]
#       ?x[owl:onProperty->?p]
#       ?x[owl:onClass->owl:Thing]
#       ?u[rdf:type->?x]
#       ?u[?p->?y]  ))

#ELIMINATED
#(* <#cls-maxqc3> *)
#Forall ?p ?y2 ?c ?u ?x ?y1 (
#   ?y1[owl:sameAs->?y2] :- And(
#       ?x[owl:maxQualifiedCardinality->1]
#       ?x[owl:onProperty->?p]
#       ?x[owl:onClass->?c]
#       ?u[rdf:type->?x]
#       ?u[?p->?y1]
#       ?y1[rdf:type->?c]
#       ?u[?p->?y2]
#       ?y2[rdf:type->?c]  ))

#ELIMINATED
#(* <#cls-maxqc4> *)
#Forall ?p ?y2 ?u ?x ?y1 (
#   ?y1[owl:sameAs->?y2] :- And(
#       ?x[owl:maxQualifiedCardinality->1]
#       ?x[owl:onProperty->?p]
#       ?x[owl:onClass->owl:Thing]
#       ?u[rdf:type->?x]
#       ?u[?p->?y1]
#       ?u[?p->?y2]  ))

#SPLIT
(* <#cax-sco> *)
Forall ?x ?c1 ?c2 (
   ?x[rdf:type->?c2] :- And(
       ?c1[rdfs:subClassOf->?c2]
       ?x[rdf:type->?c1]
             {NOTIN}{$MC} ?c2{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?c1 ?c2 (
#   ?x[rdf:type->?c2] :- And(
#       ?c1[rdfs:subClassOf->?c2]
#       ?x[rdf:type->?c1]
#             {IN}{$MC} ?c2{/IN}  ))
```

```
#SPLIT
(* <#cax-eqc1> *)
Forall ?x ?c1 ?c2 (
   ?x[rdf:type->?c2] :- And(
       ?c1[owl:equivalentClass->?c2]
       ?x[rdf:type->?c1]
             {NOTIN}{$MC} ?c2{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?c1 ?c2 (
#   ?x[rdf:type->?c2] :- And(
#       ?c1[owl:equivalentClass->?c2]
#       ?x[rdf:type->?c1]
#             {IN}{$MC} ?c2{/IN}  ))

#SPLIT
(* <#cax-eqc2> *)
Forall ?x ?c1 ?c2 (
   ?x[rdf:type->?c1] :- And(
       ?c1[owl:equivalentClass->?c2]
       ?x[rdf:type->?c2]
             {NOTIN}{$MC} ?c1{/NOTIN}  ))

#ELIMINATED
#Forall ?x ?c1 ?c2 (
#   ?x[rdf:type->?c1] :- And(
#       ?c1[owl:equivalentClass->?c2]
#       ?x[rdf:type->?c2]
#             {IN}{$MC} ?c1{/IN}  ))

#ELIMINATED
#(* <#cax-dw> *)
#Forall ?x ?c1 ?c2 (
#   rif:error() :- And(
#       ?c1[owl:disjointWith->?c2]
#       ?x[rdf:type->?c1]
#       ?x[rdf:type->?c2]  ))

#ELIMINATED
#(* <#prp-npa1> *)
#Forall ?x ?i1 ?p ?i2 (
#    rif:error() :- And(
#       ?x[owl:sourceIndividual->?i1]
#       ?x[owl:assertionProperty->?p]
#       ?x[owl:targetIndividual->?i2]
#       ?i1[?p->?i2] ))
```

```
#ELIMINATED
#(* <#prp-npa2> *)
#Forall ?x ?i ?p ?lt (
#    rif:error() :- And(
#       ?x[owl:sourceIndividual->?i]
#       ?x[owl:assertionProperty->?p]
#       ?x[owl:targetValue->?lt]
#       ?i[?p->?lt] ))

#ELIMINATED
#(* <#cax-dw> *)
#Forall ?c1 ?c2 ?x (
#    rif:error() :- And(
#       ?c1[owl:disjointWith->?c2]
#       ?x[rdf:type->?c1]
#       ?x[rdf:type->?c2] ))

#ELIMINATED
#(* <#cls-com> *)
#Forall ?c1 ?c2 ?x (
#    rif:error() :- And(
#       ?c1[owl:complementOf->?c2]
#       ?x[rdf:type->?c1]
#       ?x[rdf:type->?c2] ))


(* <#eq-diff2-3> *)
 Forall ?x ?y (
     _checkDifferent(?x ?y) :- And (
         ?x[rdf:type->owl:AllDifferent]
         ?x[owl:distinctMembers->?y] ))

 Forall ?x ?y (
     _checkDifferent(?x ?y) :- And (
         ?x[rdf:type->owl:AllDifferent]
         ?x[owl:members->?y] ))

 Forall ?x ?y ?z (
     _checkDifferent(?x ?y) :- And (
         _checkDifferent(?x ?z)
         ?z[rdf:rest->?y]
         Not(?y = rdf:nil) ))

 Forall ?x ?y1 ?y2 ?z1 ?z2 (
   rif:error() :- And (
             _checkDifferent(?x ?y1)
```

```
            _checkDifferent(?x ?y2)
            Not(?y1 = ?y2)
            ?y1[rdf:first->?z1]
            ?y2[rdf:first->?z2]
       ?z1[owl:sameAs->?z2] ))

#AUXILIARY ELIMINATION
#(* <#prp-adp> *)
# Forall ?r ?l (
#     _checkDisjointProperties(?r ?l) :- And (
#         ?r[rdf:type->owl:AllDisjointProperties]
#         ?r[owl:members -> ?l] ))

#AUXILIARY ELIMINATION
# Forall ?r ?l ?x (
#     _checkDisjointProperties(?r ?l) :- And (
#         _checkDisjointProperties(?r ?x)
#         ?x[rdf:rest->?l]
#         Not(?l = rdf:nil) ))

#ELIMINATED
# Forall ?x ?y ?o ?v ?l1 ?l2 ?r (
#   rif:error() :- And (
#             _checkDisjointProperties(?r ?l1)
#             _checkDisjointProperties(?r ?l2)
#             Not(?l1 = ?l2)
#             ?l1[rdf:first->?x]
#             ?l2[rdf:first->?y]
#             ?o[?x->?v]
#             ?o[?y->?v] ))

#AUXILIARY ELIMINATION
#(* <#cax-adc> *)
# Forall ?r ?l (
#   _checkDisjointClasses(?r ?l) :- And (
#         ?r[rdf:type -> owl:AllDisjointClasses]
#         ?r[owl:members -> ?l] ))

#AUXILIARY ELIMINATION
# Forall ?r ?l ?x (
#     _checkDisjointClasses(?r ?l) :- And (
#         _checkDisjointClasses(?r ?x)
#         ?x[rdf:rest->?l]
#         Not(?l = rdf:nil) ))

#ELIMINATED
# Forall ?x ?y ?o ?l1 ?l2 ?r (
```

```
#   rif:error() :- And (
#             _checkDisjointClasses(?r ?l1)
#             _checkDisjointClasses(?r ?l2)
#             Not(?l1 = ?l2)
#             ?l1[rdf:first->?x]
#             ?l2[rdf:first->?y]
#             ?o[rdf:type->?x]
#             ?o[rdf:type->?y] ))

#AUXILIARY ELIMINATION
#(* <#prp-spo2> *)
# Forall ?p ?pc (
#     _markCheckChain(?p ?pc) :- ?p[owl:propertyChainAxiom->?pc] )

#AUXILIARY ELIMINATION
# Forall ?p ?pc (
#     _markCheckChain(?p ?pc) :- And (
#         _markCheckChain(?p ?q)
#         ?q[rdf:rest->?pc]
#         Not(?pc = rdf:nil) ))

#ELIMINATED
# Forall ?q ?start ?pc ?last ?p (
#   _checkChain(?q ?start ?pc  ?last) :- And (
#             _markCheckChain(?q ?pc)
#             ?pc[rdf:first->?p]
#             ?pc[rdf:rest->rdf:nil]
#       ?start[?p->?last] ))

#ELIMINATED
# Forall ?q ?start ?pc ?last ?p ?tl (
#   _checkChain(?q ?start ?pc  ?last) :- And (
#       ?pc[rdf:first->?p]
#        ?pc[rdf:rest->?tl]
#       ?start[?p->?next]
#       _checkChain(?q ?next ?tl ?last) ))

#SPLIT
#AUXILIARY ELIMINATION
# Forall ?p ?last ?pc ?start (
#   ?start[?p->?last] :- And (
#       ?p[owl:propertyChainAxiom->?pc]
#       _checkChain(?p ?start ?pc ?last)
#             {NOTIN}{$MPT} ?p{/NOTIN} ))

#AUXILIARY ELIMINATION
# Forall ?p ?last ?pc ?start (
```

```
#   ?start[?p->?last] :- And (
#       ?p[owl:propertyChainAxiom->?pc]
#       _checkChain(?p ?start ?pc ?last)
#             {IN}{$MP} ?p{/IN} ))

#AUXILIARY ELIMINATION
# Forall ?p ?last ?pc ?start (
#   ?start[rdf:type->?last] :- And (
#       rdf:type[owl:propertyChainAxiom->?pc]
#       _checkChain(rdf:type ?start ?pc ?last)
#             {NOTIN}{$MC} ?last{/NOTIN} ))

##AUXILIARY ELIMINATION
# Forall ?p ?last ?pc ?start (
#   ?start[rdf:type->?last] :- And (
#       rdf:type[owl:propertyChainAxiom->?pc]
#       _checkChain(rdf:type ?start ?pc ?last)
#             {IN}{$MC} ?last{/IN} ))

#ELIMINATED
#(* <#cls-int1> *)
# Forall ?c ?l ?y ?ty (
#   _allTypes(?c ?l ?y) :- And (
#         _markAllTypes(?c ?l)
#         ?l[rdf:first->?ty]
#         ?l[rdf:rest->rdf:nil]
#         ?y[rdf:type->?ty] ))

#ELIMINATED
# Forall ?c ?l ?y ?ty ?tl (
#   _allTypes(?c ?l ?y) :- And (
#     ?l[rdf:first->?ty]
#      ?l[rdf:rest->?tl]
#     ?y[rdf:type->?ty]
#     _allTypes(?c ?tl ?y) ))

#SPLIT
#AUXILIARY ELIMINATION
# Forall ?y ?c ?l (
#   ?y[rdf:type->?c] :- And (
#     ?c[owl:intersectionOf->?l]
#     _allTypes(?c ?l ?y)
#         {NOTIN}{$MC} ?c{/NOTIN} ))

#AUXILIARY ELIMINATION
# Forall ?y ?c ?l (
#   ?y[rdf:type->?c] :- And (
```

```
#     ?c[owl:intersectionOf->?l]
#     _allTypes(?c ?l ?y)
#         {IN}{$MC} ?c{/IN} ))

#AUXILIARY ELIMINATION
#(* <#prp-key> *)
# Forall ?c ?u (
#   _markSameKey(?c ?u) :- ?c[owl:hasKey->?u] )

#AUXILIARY ELIMINATION
# Forall ?c ?u ?v (
#   _markSameKey(?c ?u) :- And (
#     _markSameKey(?c ?v)
#         ?v[rdf:rest->?u]
#         Not(?u = rdf:nil) ))

#ELIMINATED
# Forall ?c ?u ?x ?y (
#   _sameKey(?c ?u ?x ?y) :- And (
#     _markSameKey(?c ?u)
#     ?u[rdf:first->?key]
#     ?u[rdf:rest->rdf:nil]
#     ?x[?key->?v] ?y[?key->?v] ))

#ELIMINATED
# Forall ?c ?u ?x ?y (
#   _sameKey(?c ?u ?x ?y) :- And (
#     ?u[rdf:first->?key]
#     ?u[rdf:rest->?tl]
#     ?x[?key->?v] ?y[?key->?v]
#     _sameKey(?c ?tl ?x ?y) ))

#ELIMINATED
# Forall ?x ?y ?c ?u (
#   ?x[owl:sameAs->?y] :- And (
#     ?c[owl:hasKey->?u]  ?x[rdf:type->?c]  ?y[rdf:type->?c]
#     _sameKey(?c ?u ?x ?y) ))

#SPLIT
(* <#cls-uni> *)
 Forall ?y ?c ?l ?ci (
   ?y[rdf:type->?c] :- And (
         _checkUnionOf(?c ?l)
         ?l[rdf:first->?ci]
         ?y[rdf:type->?ci]
         {NOTIN}{$MC} ?c{/NOTIN} ))
```

```
#ELIMINATED
# Forall ?y ?c ?l ?ci (
#   ?y[rdf:type->?c] :- And (
#         _checkUnionOf(?c ?l)
#         ?l[rdf:first->?ci]
#        ?y[rdf:type->?ci]
#         {IN}{$MC} ?c{/IN} ))

(* <#cls-oo> *)
 Forall ?c ?l (
     _checkOneOf(?c ?l) :- ?c[owl:oneOf->?l] )

 Forall ?c ?l ?r (
   _checkOneOf(?c ?l) :- And (
         _checkOneOf(?c ?r)
         ?r[rdf:rest->?l]
         Not(?l = rdf:nil) ))

#SPLIT
 Forall ?yi ?c ?l (
   ?yi[rdf:type->?c] :- And (
         _checkOneOf(?c ?l)
         ?l[rdf:first->?yi]
         {NOTIN}{$MC} ?c{/NOTIN} ))

 Forall ?yi ?c ?l (
   ?yi[rdf:type->?c] :- And (
         _checkOneOf(?c ?l)
         ?l[rdf:first->?yi]
         {IN}{$MC} ?c{/IN} ))

#SPLIT
(* <#cls-int2> *)
 Forall ?y ?c ?ci ?l (
   ?y[rdf:type->?ci] :- And (
         _markAllTypes(?c ?l)
         ?l[rdf:first->?ci]
         ?y[rdf:type->?c]
         {NOTIN}{$MC} ?ci{/NOTIN} ))

#ELIMINATED
# Forall ?y ?c ?ci ?l (
#   ?y[rdf:type->?ci] :- And (
#         _markAllTypes(?c ?l)
#         ?l[rdf:first->?ci]
#        ?y[rdf:type->?c]
#         {IN}{$MC} ?ci{/IN} ))
```

```
#EOF
```