A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases

Ramakrishna Soma
Computer Science Department
University of Southern California
Los Angeles, CA 90089
rsoma@usc.edu

Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
prasanna@usc.edu

April 4, 2008

Abstract

Materialized knowledge bases perform inferencing when data is loaded into them, so that answering queries is reduced to simple lookup and thus are faster. A major bottleneck of such a system is the inferencing process, which is slow and memory intensive. In this work we examine the problem of speeding up as well as scaling the inferencing process for OWL Knowledge Bases, that employ rule based reasoners. We propose a data partitioning approach for this problem, in which the input data is partitioned into smaller chunks that are then processed independently. We propose a parallel reasoning algorithm and show the correctness of our technique for the class of rule-sets obtained from OWL ontologies. We present two optimizations of this algorithm to further improve the performance. Finally, we present an implementation based on a popular open source tool. In our experiments using a standard OWL benchmark we have observed speedups of upto 18x, on a parallel cluster of 16 processors.

1 Introduction

The semantic web standards have been proposed as the next generation approach to managing and searching for data on the internet. *Ontologies* and *knowledge bases* are two important components of a semantic web system. Ontologies are schemas that encode a reality in a machine interpretable way; knowledge bases are databases which are based on this schema. Two key standards in the semantic web stack are RDF(S) and OWL, which are ontology representation languages. OWL, which is the focus of this paper, is based on Description Logics (DL), a well studied subject in the AI community [2]. An important, advantage of using OWL ontologies is the ability to perform automated reasoning on the data instances based on the ontology

definition (and defined by Description Logic semantics). For the semantic web to be successful, scalable methods to perform this reasoning are important.

Reasoning can be performed either when the data is loaded into the knowledge base (KB) or when a query is issued. The former class of KBs, are called materialized knowledge bases and are the focus of this paper. Materialized KBs trade-off space and increased loading time for shorter query times, because no reasoning is performed during query answering. This approach is suited for applications domains where the frequency of data addition is much lesser than that of running queries. Examples of such applications are data warehouses and (for most part) web-search. Since the worst case complexity for OWL reasoning is exponential in time and memory [2], this approach is often considered to be a good way to store OWL KBs.

Most reasoning engines for OWL are implemented using either tableau algorithms or rule based/logic programming based engines. The OWL reasoners that are implemented using rule based engines, have been suggested as a practical alternative to the correct and complete tableau algorithms [8]. The main advantages of this class of reasoners are that they are well studied and many robust implementations exist. The disadvantages are that only a subset of the least expressive fragment of OWL (Lite) can be implemented using these.

In this paper we consider the semantics of OWL, called OWL-Horst, which is defined in [15]. The semantics of OWL-Horst diverges from the standard in a few ways: it only covers a subset of OWL constructs, it specifies if based semantics as opposed to the iff based semantics in the OWL specification and finally, unlike OWL-Lite or OWL-DL, it is fully compatible with RDFS- especially the notion of using a class as an instance. Interestingly, the semantics of the dialect, are presented as a set of simple rules that can be implemented using datalog semantics. Many of the currently available semantic-web tools, both open-source (Jena) and commercial (OWLIM, Oracle) implement the OWL-Horst rule-set or variants/extensions of it, using rule-based systems.

Parallel algorithms have been used to improve the scalability and performance of algorithms. To parallelize a task, it's *workload* is divided into sub-parts. Each of these sub-parts, is processed independently by a processor, communicating with others when necessary. For the rule based reasoning task, the *workload* can be partitioned by either partitioning the *rule-set*, or partitioning the *data*, or partitioning both of them. In this paper, we have used the data partitioning approach. Intuitively, since, the datasets are much larger than the rules, in the semantic web applications we consider, this approach is well suited for the problem. The following are the main contributions of our work:

- We propose a data partitioning approach and a parallel algorithm for scalable and faster OWL reasoning. We also propose

two optimizations to the basic parallel algorithm.

- We present an implementation of our algorithm based on a open source rule-based reasoner and present experimiental results for it. To the best of our knowledge, ours is the first implementation of a parallel processing algorithm for OWL inferencing.

The state-of-the art technology addressing this problem has demonstrated for relatively small data-sets (1 Billion triples) and for simple rule-sets. This is by no means web-scale, which is likely to be a few orders of magnitude larger both in terms of data-set size and complexity of the rule-sets. Parallel inferencing, in our opinion, is a promising approach to scale the computations to such large volumes.

The rest of the paper is organized as follows. First, we provide a classification of rule-sets and their impact on data partitioning. We then describe the parallel algorithm and prove its correctness. We describe our implementation and finally, we present some of our results and conclusions.

2 Rule Classes and OWL-Horst

In this section we introduce a classification of datalog rules based on the organization of the clauses in the body of the rules. The relevance of such a classification is that, each class of rules, affords different levels of ease with which the base tuples can be partitioned. The datalog semantics we consider in this paper are that of the negation free conjunctive datalog as applied to RDF triples.

- **Single sub-goal rules:** These are the simplest kind of rules, which have only one sub-goal in the body. Simple class and property hierarchies can be expressed as single sub-goal rules. E.g., the rule shown below, states that if a resource is of type person, it is also of type OWL Thing class.

  R1: *?X rdf:type lubm:Person* $\rightarrow$ *?X rdf:type owl:Thing*

  If a rule-set only contains single sub-goal rules, the base-triples can be arbitrarily partitioned and each partition processed independently.

- **Chained sub-goal rules:** These are rules which have multiples sub-goals, but with the constraint that at least one of the variables in every sub-goal is joined with a variable in some other sub-goal. E.g., the rule shown below defines the

owl:differentFrom predicate based on two classes defined to be disjoint. Here the first sub-goal and the second sub-goal share a (join) variable and the three sub-goals share two join variables ?C and ?D:

R2: *(?A rdf:type ?C) (?C owl:disjoint ?D) (?B rdf:type ?D) →?A owl:differentFrom ?B*

Depending on the number of sub-goals in the rule, chained sub-goals can be further categorized into:

– **Single-join rules:** Chained sub-goals which only have two sub-goals and therefore share at least one (join) variable are called single-join rules. E.g., the recursive rule shown below, is used to implement a OWL transitive property. The join variable B is shared between the two sub-goals of the rule.

R3: *(?A lubm:SO ?B) (?B lubm:SO ?C) →?A lubm:SO ?C*

Note that, every single join rule can be correctly re-written as a single-join rule by repeating the sub-goal in the body two times. E.g., rule R1 is re-written as follows in a single-join rule form:

R1': *(?X rdf:type lubm:Person) (?X rdf:type lubm:Person) →?X rdf:type owl:Thing*

Data partitioning for a rule-base that only contains single join rules is harder than partitioning for single sub-goal rules because, we must ensure that, each pair of tuples that can fire a rule must be present in the same partition. This can be achieved by a partitioning strategy that uses a static hash function mapping a tuple onto a processor based on its subject or object properties. All the tuples which could produce an inferred triple, have a join variable in common, and will be hashed on to the same partition.

– **Multi-join rules:** Chained rules that contain (n $>$ 2) sub-goals, and hence at least (n-1) join variables are called multi-variable join rules. The *differentFrom* rule R3, given above is an example of such a rule, containing three sub-goals and two join variables. Data partitioning for a multi-join rule base is harder than single-join rule-base because the static hash function will need to partition a larger combinatorial space of resources. E.g, for a two-join rule and a data set consisting of n resources, the n^2 space of the possible combination of resources must be partitioned such that for a certain pair of join values, all the tuples that can potentially be joined through them are available on the same processor. The problem is harder when we have to make sure that the partitions need to have the equal workloads.

- **Independent sub-goal rules:** These are rules containing more than one sub-goal such that, the sub-goals do not share any variables. E.g., in the rule shown below, the two subgoals do not share any variables.

R4: *(?A P1 ?B) (?C P2 ?D)* $\rightarrow$*?A P1 ?D*

Partitioning a data-set for a rule-set consisting of independent sub-goals can be complex because, each rule stipulates a different partitioning of the space and hence the partitioning function can be very complex.

All but one of the rules from OWL-Horst can be written as single-join rules, when the T-Box is known before-hand and no new blank nodes are introduced into the RDF graph. The one rule that cannot be written as a single-join rule is a two-join rule.

3 Parallel reasoning for OWL-Horst knowledge-bases

The first step of the parallelization algorithm is to partition the data set. Various algorithms can be used to partition the data as long as they have the following two characteristics.

1. Lossless: Each triple must be in at-least one partition.
2. Join Co-location: Every triple which can be potentially joined must be present in the same partition. This means that every triple that has the same subject, object or predicate values must be present in the same partition. The rule-sets for OWL can be written such that the predicate of each sub-goal in the body of the rules, is always a constant. Therefore, the join co-location rule stipulates that if there are two tuples with same subject or object, they should be present in the same partition.

We will call partitioning algorithms that have these characteristics as single-join correct algorithms and a partition created by such a algorithm single-join correct partition. A generic initial partition algorithm is given in algorithm 1. The algorithm uses an abstract partitioning function to create a partition table (also called owners list) that assigns each resource to a partition.

3.1 Parallel inferencing algorithm

An algorithm for parallel reasoning is presented in algorithm 2. The algorithm receives an initial single-join correct partition consisting of p partitions and applies the same rule-set (datalog program) identically to each partition to reach the

Algorithm 1 Partition data-set

Input: Initial tuples

Output: Set of partitions of original tuples, partition table

1: Partition the resulting graph based on the partitioning policy. Create a *owners list* assigning a resource to a partition.

2: **for** every tuple **do**

3: Assign the tuple to partition that is owner of the subject and the partition that is the owner of the object of the tuple

4: **end for**

5: **return**

least-fix point (LFP) for the partition. We refer to each application of the rule-set to obtain the LFP as a round. The algorithm transmits a new tuple created in a particular round at a certain processor, to the owner of its subject or object. At the beginning of the each round, all the tuples transmitted to a processor are gathered and the LFP for the new model is generated. The algorithm terminates, when each processor of the system has finished a round without creating tuples that need to be communicated to another processor and there are no tuples in transit. This is very similar to the algorithm in [7], which proves some of the correctness results for this algorithm.

Algorithm 2 Parallel Reasoning

Input: Initial base tuples

Output: Base tuples and Inferred tuples

1: Call **Partition**, to obtain initial partition and the owner list for each processor

2: Send the owner list to all processors

At each Processor:

3: **while** !terminate **do**

4: Generate least fix-point model of the current data set

5: Send newly generated tuples to other processors as in **Partition**

6: Receive tuples from other processors add them to the base tuples.

7: **end while**

8: **return**

3.2 Correctness

The correctness of the algorithm is asserted in the following lemma.

Lemma 1. *Every tuple that is present in a model generated by a serial reasoner is also present in a model generated by the above algorithm, for a single-join rule set with known universe of atoms and reasoning defined by negation free datalog semantics when the partition is produced by a single-join correct partition.*

Proof sketch: Intuitively, our algorithm ensures that any two tuples that can be joined to form a new tuple are present in the same partition. This applies to tuples that are present in the base partition as well as those that are generated in the later rounds of the algorithm. Thus, any tuple that is derived in the serial algorithm is also derived in the parallel algorithm and the resulting model is bound to be the same. The complete proof is presented in [1].

4 Implementation And Optimizations

We have used the Jena package [11] as the base reasoning platform. Jena uses a *hybrid* reasoner, which uses both the forward and backward chaining methods. The forward chaining is implemented using the Rete algorithm where as the backward chaining is implemented using the standard xsd resolution with support for tabling. In general, the hybrid engine works by first *compiling* the ontology into rules by using the forward engine. These rules are used by the backward engine to derive new tuples. To materialize a KB, a query of the form *for each resource, select all triples from the KB with that resource as subject* is issued. This triggers the reasoner and generates the inferred tuples in the KB.

Jena is used in our work because, it is a mature reasoner, is freely available, and implements the dialect of OWL that we have used in our work. We intend to implement our techniques using a different reasoner in our future work. Although, a different reasoning strategy can be used, e.g., bottom-up datalog evaluation, the contributions of our work is not diminished because, our work is applicable to any kind of reasoner that adheres to datalog semantics.

4.1 Partitioning

Apart from the correctness conditions for the partitioning algorithms specified earlier, the other important goal of the partitioning step is to divide the workload among the processors so that each partition, can execute its own, reduced workload, communicating with other processors if necessary. The main goals of partitioning the data are to achieve:

1. Balanced partitioning: The amount of work done on each processor should be nearly the same.

2. Minimize communication: Data to be transferred between processors should be minimized.

3. Efficiency: Ideally each triple in the result must be derived by exactly one processor. In general, we want to minimize the number of time inferences are duplicated in the processors.

4. Speed and Scalability: The partitioning itself should be fast and scale for extremely large data-sets.

The above problem can be formulated as a **graph partitioning** problem. In the classical weighted/multi-constraint graph partitioning problem, the input is a graph G={V, E, W}, where V is the set of vertices, E is the set of edges and W is a function that assigns weights to vertices. The goal is to divide G into *k* sub-graphs, such that, the sum of weights of verices in each sub-graph is nearly the same while the number of edges cut, i.e., the edges between nodes in different partitions is minimized. The input RDF graph, in which, each triple is represented by two vertices, one each for the subject and the object and an edge representing the property is considered for partition. All the vertices and edges are uniformly weighted. The result of the partitioning stage are the *k* sub-graphs. The set of vertices in each partition form the owner list for a partition. For every edge cut in the partitioning, the subject and the object of a triple are *owned* by the different processor. This is illustrated in the figure 1. Here the inital graph is partitioned, such that {A,D,E} belong to partition P1 and {B,C,F} belong to another partition. Since the edge BE is cut by the partition, the vertex B (E) is replicated in P1 (P2) and the edge BE is replicated on both nodes.

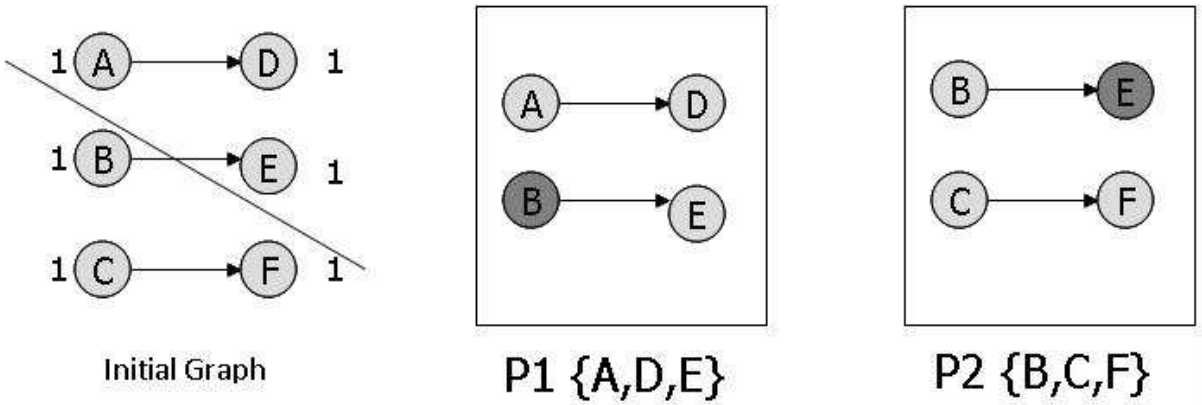

Figure 1: Illustration of the partitioning algorithm.

For datalog processing the (worst case) complexity of the inferencing is $O(n^m)$, where m is the maximum number of free variables present in the body of any rule and n is the number of constants in the Herbrand base i.e. nodes in the input graph.

Since the rule-set used in each of the partitions is the same, the load can be partitioned by dividing the number of nodes equally among the processors. This satisfies one of the goals of graph partitioning problem: to ensure that the number of vertices in each partition are equal, this goal. Further, by minimizing the edge-cut of the input graph, we address both efficiency and minimum communication. A statement `R1 P R2` can be derived in one partition only if both `R1` and `R2` are present in the same partition. Since minimizing edge cut also minimizes the number of vertices that are replicated in partitions, efficiency is addressed. Similarly a tuple `R1 P R2` is communicated iff `R1` and `R2` are owned by different partitions.

Although the graph partitioning problem is a NP-complete problem, many heuristics have been proposed which are fast and scalable [6]. The graph partitioning package that we use in our implementation, called Metis, provides very efficient implementations of algorithms which run in O(V) time where V is the number of vertices in the graph [12].

4.2 Optimization 1: Incremental addition algorithm

An important challenge of the implementation is to ensure that the later rounds of reasoning, when only a few tuples are added to the data-set, are executed efficiently. A naive approach would query the reasoner for all the tuples in the knowledge base- which would cause the reasoner to perform many of the derivations again. Our implementation, instead only considers the statements that are added to the partition in the current round and all the statements that may in turn be created due to their addition. The algorithm 3 is presented below and is similar in spirit to the work done on incremental view maintainence algorithms [10].

Algorithm 3 Incremental addition algorithm

Input: oldTriples, addedTriples

Output: inferredTriples

1: Initialize *resource list* as the all resources that appear as either the subject or object in *addedTriples*
2: **for all** resources in *resource list* **do**
3: Query for all statements which have the subject or object as the current resource
4: **for all** new statements **do**
5: if the subject or object does not already appear in the resource list add it to the list
6: **end for**
7: **end for**
8: **return**

Our incremental addition algorithm is based on the observation that the OWL rules are *range restricted* i.e. all the variables that appear in the head of the rule also appear in the body of the rule. Moreover, the subject term in every rule is always a variable. The correctness of this algorithm is stated in the following lemma.

Lemma 2. *A statement appears in the derived model iff the statement appears in the LFP of a kb which contains the union of the old and added triples.*

Proof. →Let us assume that there is a tuple in the LFP of the model consisting of the union of input and added triples that is not present in the model that we derived. Obviously, the subject and the object of such a tuple is not one of the resources that appears as a subject or object in a triple of one of the newly added triples- otherwise it would have been generated by our algorithm. Since all our rules are join rules, this tuple should have been generated due by a rule that was matched by tuples only in the model from the previous iteration. But such a tuple would have already been generated by a previous iteration and hence exists in the model generated by our algorithm. Thus, every tuple that is in the LFP of the union of the kb from the previous iteration and the newly added tuples is also present in the model generated by our algorithm.

←Because our algorithm uses a sound datalog reasoner, by definition, every tuple generated by our algorithm has to be in the LFP of the union of the old model and the newly add tuples. Thus the two models are equivalent.

□

4.3 Optimization 2: T-Box exclusion

This optimization is based on the observation that in OWL-DL and OWL-Lite reasoning, facts in the A-Box do not add any facts about elements in the T-Box. Thus, the facts from the T-Box can be shared/replicated among all the nodes in the partition and during partitioning, these nodes are excluded. Since, a large number of facts in a typical OWL graph are type assertions, the number of edges incident on a T-Box element are very high. If such nodes are not excluded during partitioning, the node that is the owner of a T-Box element tends to have a larger number of statements in it than other nodes. This results in a partition in which one of the parts has a much larger processing time than others. By excluding these nodes during partitioning, the partitions obtained are more balanced. This optimization does not hold for RDF, OWL-Full as well as the full OWL-Horst reasoning, where there is no separation between the A-Box and the T-Box.

5 Experimental Results

To measure the performance of our algorithm, we conducted experiments on a cluster of machines at HPCC in USC. The experiments were conducted on AMD Opteron 2.6 GHz, 64-bit dual core machines, with main memories between 4-64GB. Each partition was executed on a separate processor core. Inter-partition communication was achieved through shared files: each partition at the end of a round, wrote the tuples to be communicated to a processor, in a specific file. This file was then read by the destination partition at the beginning of each round of reasoning. Currently, we have run the reasoning for the LUBM(10) (1M triples) and UOBM(5) data-sets. The speed-up for this is shown below in figure 2.

The results somewhat surprisingly show a super-linear speedup for the LUBM data-set. The reason we see this is because the Jena reasoner implements a theorem prover that uses SLD-resolution, which can be thought of as a Depth First Search (DFS) on the AND-OR tree [4]. During partitioning, the search space for the theorem prover is decreased for some data/rule-sets. Thus, in spite of the replication in the input tuples, the reasoner performs lesser operations, thus giving us the super-linear speedup. On the other hand, the results are more modest for the UOBM data-set- we have obtained speedups of about 4 on a 16 node processor. This is because, the UOBM graph is more interconnected, thus leading to more replication and the rule-set does lead to reduced search spaces.

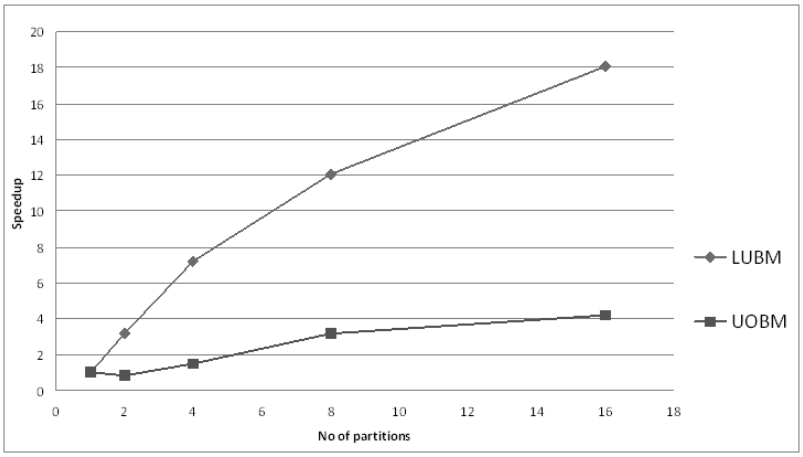

Figure 2: Speedup for the LUBM-10 benchmark on different number of processors.

More densely connected graphs often occuring due to symmetric and transitive properties (including owl:sameAs and differentFrom) are less amenable to parallelization. In such cases, it might be a good idea to perform *partial* materialization, where specific reasoning is not performed at materialization time but rather at query time and the rest of the reasoning can be parallelized.

5.1 Decreasing memory footprint/Scalability

Another problem of semantic web reasoning is that the reasoning is memory hungry. We have used our data partitioning approach to decrease the maximum memory footprint of the reasoning process by processing one partition at a time on a processor. The results in figure 3 show the relative time taken and the memory foot-print for the LUBM-10 benchmark.

From the graph above, we see that even if we process a partitioned model serially, the reasoning executes faster. At the same time, the memory foot-print of the application shrinks. For the case with 16 partitions, the memory foot-print is about 15% of the serial reasoner where as the time taken is about half of it.

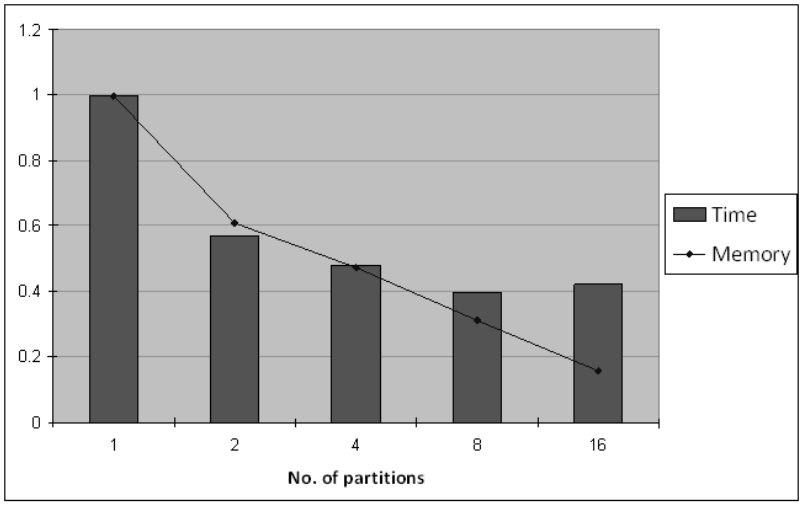

Figure 3: Time vs. Memory trade-off for the LUBM benchmark.

5.2 Incremental addition

Figure 4 demonstrates the advantage of using our incremental addition algorithm. We see that without the incremental addition algorithm, the time taken for second iteration is more than half that of the first iteration. Since, Jena implements a tabled top-down resolution, which avoids multiple re-derivations and hence, the time taken is lesser than iteration one. In the second figure we see that iteration 2 takes about 1 percent of the time for iteration 1.

6 Related Work

The problem of scaling and speeding up OWL reasoning is an exciting area of active research. Most current approaches work on scaling *serial* reasoners. OWLIM [14] uses a rule based forward chaining reasoner called TREEE that achieves efficiency and speed by converting rules into chunks of Java code together with an efficient in-memory representation of the OWL graph. Oracle [13] implements a forward-chaining rule-set in SQL, and performs the reasoning with in the database.

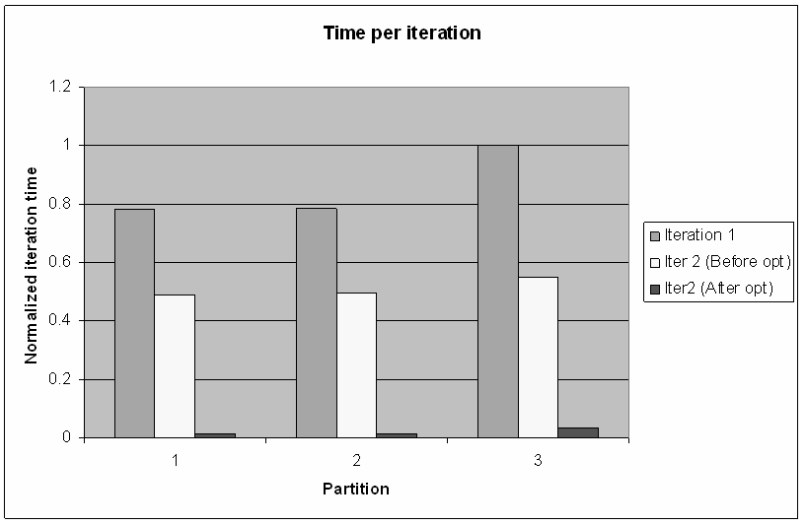

Figure 4: Time spent per iteration on each partition.

Thus, they leverage much of their work in optimizing their database to achieve very good performance for OWL reasoning. Since our approach is based on datalog semantics, and the rule-sets use in the two reasoners mentioned above are similar to what we have studied, our partitioning technique can be applied for these reasoners.

Much work has been done in the area of parallelizing rule based systems, although to the best of our knowledge ours is the first work to examine the problem of parallelizing rule-sets for OWL ontologies. The work on parallelizing rule systems can be classified based on how the inferencing workload is partitioned. In rule-base partitioning, the set of rules are partitioned and each processor works on the original dataset (base tuples) but using a subset of the original rule-base [3, 9]. This kind of partitioning is best suited for systems where the rule-sets are much larger than the datasets, e.g., expert systems or when the dataset cannot be readily partitioned. Finally, in a *hybrid partitioning* approach the rule-set is first partitioned and based on this the data is then partitioned. An example of such work for OWL KBs is presented in [5]. In contrast to these works, we have explored the data partitioning approach in our work.

Our work is most similar to and builds on the algorithms to parallelize datalog programs by data partitioning, presented in [7, 17, 16]. The algorithm used for parallelizing datalog programs is very similar to the ones presented earlier. These works use an abstract hash function that is used to determine the partitioning of a dataset. In our work we examine the problem as applied to the rule-sets generated by OWL ontologies as opposed the more general treatise in these works. We have exploited the fact that OWL rule-sets are limited to a simple class of datalog rules, to create a simple but effective partitioning mechanism for the datasets.

7 Conclusions

Our partitioning algorithm is based on the observation that the rule-set generated for the OWL-Horst ontologies, use only single-join rules. This enables us to use a fairly simle technique to partition the datasets and to process them in parallel. The ability to partition the datasets and process them in parallel enables us to both speedup the processing time by employing many processors in parallel. Our approach can be used not only on parallel clusters, but also on the popular multi-core processors in the market today. It also enables us to reduce the memory footprint of the reasoning process because only a subset of the data needs to be loaded into the memory at a given time. The results we have obtained on standard benchmarks are promising with speedups upto 18x being observed on a 16 node parallel cluster.

Acknowledgment

This research was funded by CiSoft (Center for Interactive Smart Oilfield Technologies), a Center of Research Excellence and Academic Training and a joint venture between the University of Southern California and Chevron. We are grateful to the management of CiSoft and Chevron for permission to present this work. Computation for the work described in this paper was supported by the University of Southern California Center for High-Performance Computing and Communications (www.usc.edu/hpcc).

References

[1] Proof for lemma 1, http://www-scf.usc.edu/ rsoma/parallel/proof.pdf.

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

[3] D. A. Bell, J. Shao, and M. E. C. Hull. A pipelined strategy for processing recursive queries in parallel. *Data Knowl. Eng.*, 6:367–391, 1991.

[4] P. Boizumault. *The Implementation of Prolog*. Princeton series in Computer Science, 1993.

[5] J. Du and Y.-D. Shen. Partitioning aboxes based on converting dl to plain datalog. In *The Twentyth International Workshop on Description Logics (DL-07),*, 2007.

[6] U. Elsner. Graph partitioning - a survey, 1997.

[7] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In *SIGMOD Conference*, pages 143–152, 1990.

[8] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW*, pages 48–57, 2003.

[9] A. Gupta, C. Forgy, A. Newell, and R. G. Wedig. Parallel algorithms and architectures for rule-based systems. In *ISCA*, pages 28–37, 1986.

[10] A. Gupta and I. Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, Cambridge, MA, USA.

[11] Jena semantic web framework, http://jena.sourceforge.net/.

[12] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing,*, 1998.

[13] Oracle semantic technologies, http://www.oracle.com/technology/tech/semantic_technologies/index.html.

[14] OWLIM semantic repository, http://www.ontotext.com/owlim/index.html/.

[15] H. J. ter Horst. Combining rdf and part of owl with rules: Semantics, decidability, complexity. In *International Semantic Web Conference*, pages 668–684, 2005.

[16] O. Wolfson and A. Ozeri. Parallel and distributed processing of rules by data reduction. *IEEE Trans. Knowl. Data Eng.*, 5(3):523–530, 1993.

[17] W. Zhang, K. Wang, and S.-C. Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. Knowl. Data Eng.*, 7(1):163–176, 1995.