

Parallel Inferencing for OWL Knowledge Bases

Ramakrishna Soma

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

V.K.Prasanna

Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: prasanna@usc.edu

Abstract—We examine the problem of parallelizing the inferencing process for OWL knowledge-bases. A key challenge in this problem is partitioning the computational workload of this process to minimize duplication of computation and the amount of data communicated among processors. We investigate two approaches to address this challenge. In the *data partitioning* approach, the data-set is partitioned into smaller units, which are then processed independently. In the *rule partitioning* approach the rule-base is partitioned and the smaller rule-bases are applied to the complete data set. We present various algorithms for the partitioning and analyze their advantages and disadvantages. A parallel inferencing algorithm is presented which uses the partitions that are created by the two approaches. We then present an implementation based on a popular open source OWL reasoner and on a networked cluster. Our experimental results show significant speedups for some popular benchmarks, thus making this a promising approach.

I. INTRODUCTION

The semantic web is a vision of the World Wide Web in which the content is easier to find, share, and integrate. An important step in realizing such a vision is the ability to represent the content in a richer and machine interpretable manner. To address this issue, the W3C consortium, the standards body for the internet has defined a set of standards for knowledge representation and reasoning. An important standard in this landscape is the Web Ontology Language (OWL), which is based on the Description Logic (DL) formalism. OWL provides a set of rich data modeling constructs like classes, class hierarchies, property hierarchies etc. Such features are used to define data schemas or *ontologies* for a domain, which describe entities in the domain, their properties and relationships, and constraints between them. The instance data, compliant with these ontologies are stored in *knowledge bases*. Apart from semantic web applications, OWL has also been proposed to address similar information search and integration tasks for grids and for large enterprises.

An important feature of using OWL is that based on these ontology definitions, specialized tools called reasoners can infer additional information from the information presented. As a simple example of such reasoning, a *Student* can be defined as a sub-class of a *Person* in an ontology. When, a document asserts that S_A is a *Student*, the OWL reasoner infers that S_A is a *Person*. This kind of reasoning is important for enabling intelligent applications and searches. It can be performed either when the data is loaded into the knowledge base or when a query is issued. The former class of knowledge

bases, which perform reasoning when data is loaded are called *materialized* knowledge bases and are the focus of this paper. Materialized knowledge-bases trade-off space and increased loading time for shorter query times. This approach is suited for applications domains where the frequency of data being added is much smaller than that of queries being presented. Examples of such applications are data warehouses and (for most part) web-search. Moreover, since the worst case for OWL reasoning is exponential in time and memory [1], this approach is often considered to be a good way to store and query 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 more correct and complete tableau algorithms [2]. In rule based reasoners, the OWL ontology definitions are first *compiled* into a set of rules. This rule-set is then applied on the presented data-set to create the new inferred triples. 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 OWL specification can be implemented using them. Many popular open source (Jena [3]) and commercial OWL toolkits (OWLIM [4], Oracle [5]), are implemented using rule based reasoners.

For OWL to be successful as a semantic web language, it is important that the reasoning is scalable to large ontologies and data-sets. The state-of-the-art tools have been shown to work only on simple ontologies and data-sets, which are by no means web-scale. One way to improve the scalability and performance of reasoning is by applying parallel processing techniques to the reasoning process. In this paper, we study the problem of parallel inferencing for materialized OWL knowledge bases which use rule-based reasoners. In an abstract sense, the main problem of parallelization, is to partition the computational workload of the process, into smaller sub-parts. In the rule-based OWL inferencing context, the computational workload depends on the data-set and the rule-set presented. Thus two ways to partition the workload are by partitioning the rule-base and by partitioning the data-set. We examine both these approaches in this paper. More specifically, our contributions are as follows:

- We focus on the problem of partitioning the computational workload of OWL inferencing and propose two

approaches to the problem. In the first approach, the data-set is partitioned and the complete rule-base is applied to the each subset of the data. In the second approach, the rule-base is partitioned and each node of a parallel system applies one subset of rules to the original data-set. To the best of our knowledge ours is the first work that examines the problem of partitioning in the context of parallel reasoning for rule-based OWL knowledge bases.

- We present an abstract data partitioning algorithm which is based on the observation that rules generated for an OWL ontology fall into a particular class of rules. We then propose three concrete partitioning algorithms and present metrics which can guide researchers to create and evaluate other data partitioning algorithms in the future.
- We present an implementation based on an existing open source tool on a network cluster and on multi-core machines. Experiments on a standard OWL benchmark have shown speed-ups of about 18x on a 16 node network. The super-linear speedups are observed because for some data-sets the partitioning reduces the search space that the reasoner explores, thereby reducing the overall work done.

The rest of the paper is organized as follows: in section II we review some of the background. We then present our two approaches as well as different algorithms for partitioning in section III. We then present the generic parallel reasoning algorithm in section IV and present our implementation section V. Finally, we present the results obtained on two standard OWL benchmarks with some analysis in section VI.

II. BACKGROUND

OWL provides a rich suite of features for data modeling/knowledge representation. The ability to represent classes, properties of these classes and class hierarchies is one such feature. Another important feature is the ability to define a property as a *functional* property, a *symmetric* property, *transitive* property etc. A functional property is the same as the idea of a primary key in relational databases, i.e., an object can only have one value for that property. Transitive property is best explained through an example: the `brotherOf` relationship is a transitive property, i.e., *If A is a brotherOf B and B is a brotherOf C then A is a brotherOf C.*

The OWL language specification is presented as a set of three dialects- OWL-Lite, OWL-DL and OWL-Full. Each of these has a varying degree of expressiveness and computational complexity. OWL-Lite is the least expressive dialect, easiest to implement and least expensive computationally. On the other hand OWL-Full is the most expressive but undecidable. OWL-DL falls in between the two languages in terms of expressiveness and complexity. In this paper, we use a popular but non-standard subset of the OWL specification called OWL-Horst, first presented in [6]. The semantics of OWL-Horst diverges from the standard in a few ways:

- It only covers a subset of OWL-Lite constructs- mainly it specifies if based semantics as opposed to the iff based semantics in the OWL specification.

- Unlike OWL-Lite or OWL-DL, it is fully compatible with RDFS- especially the notion of using a class as an instance.

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.

We study the parallelization of rule based implementations of OWL reasoning. In general, the rule based engines for OWL work by compiling the ontology into a set of rules. These are then applied to the data-set to create the inferred data. The semantics of the rules are defined using negation free datalog [7]. Datalog is a simplified subset of Prolog, and reasoning using datalog has polynomial time complexity. All the rules of the OWL-Horst can be written as datalog rules. The datalog rules are written as *head* \leftarrow *body* (read if *body* then *head*). The head of the rule has only one clause and the body of the rule is a horn clause with many *sub-goals*. In fact, we have observed that only a small class of rules called *single-join* rules can be used to represent all but one of the rules. Single-join rules are rules which have two sub-goals in the body of the clause and both these sub-goals share a variable. An example of such a rule is the rule used for reasoning for transitive properties:

$$(?A \text{ brotherOf } ?B) \text{ AND } (?B \text{ brotherOf } ?C) \rightarrow ?A \text{ brotherOf } ?C$$

Here the body of the rule has two sub-goals and both these sub-goals have the variable `?B` as a common variable. As will be seen later, the fact that the rules for OWL ontologies can be written using such a simple class of rules makes it easy for us to devise a parallel processing algorithm. A more thorough examination of this including the proof of correctness is outside the scope of this paper and is presented elsewhere [8].

III. PARTITIONING

Since the computational workload depends on the rule-set and the data-set, a viable partitioning scheme can partition either of them or both (hybrid). Irrespective of how the computational workload is partitioned, the partitioning scheme must achieve the following goals, so that the parallel processing is efficient:

- 1) **Balanced partitioning:** The amount of work done on each processor should be nearly the same. Otherwise, a partition may have to wait for another one to complete its processing, thereby wasting time.
- 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 times the inferences are duplicated in the processors.
- 4) **Speed and Scalability:** The partitioning itself should be fast and scale for extremely large data-sets.

A. Data Partitioning

The goal of data partitioning is to partition the data-set such that each partition processes a sub-set of the data using the complete rule-set. Our data partitioning algorithm makes use of the fact that all the rules for OWL reasoning can be written as single join rules. For reasoning on single-join rules to be correct, a data partitioning algorithm must make sure that any two tuples that can potentially join must be present on the same partition. Let us consider a simple rule which specifies a transitive property:

(?A brotherOf ?B) AND (?B brotherOf ?C) \rightarrow
?A brotherOf ?C

If there are two tuples {Adam brotherOf Bob} and {Bob brotherOf Charlie}, then both these tuples must be present in the same partition for the rule to fire successfully. To ensure this, we make sure that all tuples with Bob as a subject or an object are all present in the same node. In other words, we assign the ownership of each resource in the graph to a particular node and ensure that each tuple that contains the resource as subject as subject or object is always present on that node. Thus every rule that can be fired will be fired correctly. The generic data partitioning algorithm is presented below.

Algorithm 1 Data partitioning

Input: Initial tuples

Output: Set of partitions of original tuples, partition table)

- 1: Remove all the tuples involving the schema elements from the initial tuples.
 - 2: Partition the resulting graph based on the partitioning policy.
 - 3: **for all** tuples **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**
-

The generic data partitioning algorithm above only says that an ownership list has to be generated but does not specify how that list is generated. Various algorithms can be used to generate such a list and we have implemented the following algorithms.

1) *Graph partitioning*: 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 it into k sub-graphs, such that, the sum of weights of vertices in each sub-graph is nearly the same and 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 are uniformly weighted. The partitioning results in k sub-graphs,

from which we extract an *owner-list*, which is the set of vertices in a partition. For every edge cut in the partitioning, the subject and the object of a triple are *owned* by the different processor. Such triples are replicated on both the machines. Therefore, a triple from the dataset can be present in at most two processors.

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 partition is the same, the load can be partitioned by dividing the number of nodes equally among the processors. This requirement is satisfied by one of the goals of graph partitioning problem: to ensure that the number of vertices in each partition are equal. 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. Finally, although the graph partitioning problem is a NP-complete problem, many heuristics have been proposed which are fast and scalable [9]. The graph partitioning package that we use in our implementation, called Metis, has been shown to work for graphs with millions of nodes.

2) *Hash based partitioning*: In the hash based approach, a (generic/arbitrary) hash function is used to determine which processor a node is assigned to. The hash based approach is easy to implement and has been shown to work well for data partitioning problems in different domains [10]. The advantages of the algorithm is that it can be implemented as a streaming algorithm, i.e., the whole data graph need not be loaded into the memory for the partitioning. Moreover, if a sufficiently cheap hashing function is used, the owner-list need not be replicated in each partition, thus making for more efficient and scalable implementations. On the other hand, the disadvantage is that the hashing algorithm does not minimize edge-cuts and therefore, the replication in the partitions could be very high.

3) *Domain specific partitioning*: A domain specific partitioning algorithm, uses knowledge about the characteristics of particular data-sets to create a partitioning. As an example, the LUBM is an OWL benchmark that models concepts in a university setting. Typical concepts include: Departments, Students, Professors, Publications etc. The entities are organized such that entities that belong to a certain university are more likely to be related to each other, than entities that belong to different universities. We have used this characteristic of the data to create a partitioning algorithm. Like the hash-based algorithm, the domain specific algorithm can be implemented as a *streaming* algorithm, so that the whole input graph need not be loaded into the memory, which makes the algorithm more scalable. A well thought out partitioning scheme could lead to better partitions than the hash algorithm. Obviously, the

algorithm itself cannot be reused across data-sets and different partitioning scheme must be implemented for different data-sets.

Since a multitude of such partitioning algorithms can be devised, we need a way to determine the effectiveness of an algorithm, To this end, we propose the following metrics that address the four goals of such algorithms.

- 1) **Balanced partition:** The standard deviation of CPU time of each processor in the system is the ideal parameter for measuring balanced partitions. However, since the CPU time is not known before a parallel algorithm is run, a *diagnostic metric*, which can be used is bal , given as *standard deviation of the number of nodes in each partition*. This is a useful metric because the computational time of the reasoning is directly proportional to the number of nodes in the RDF graph.
- 2) **Efficiency:** The metric is the output replication given by, $OR = \Sigma (\text{No. of tuples in the result of each processor}) / \text{Total no. of tuples in the unioned output graph}$
A diagnostic metric is the input replication given by, $IR = \Sigma (\text{No. of nodes in each processor}) / \text{Total no. of nodes in the input graph}$
- 3) **Minimize communication:** As above since the exact number of tuples communicated cannot be determined before hand, a reasonable diagnostic metric for minimizing communication is the input replication. This is because, only the tuples that are related to the nodes that are replicated, are transmitted.
- 4) **Speed:** The time taken for the partitioning itself is used as the metric.

B. Rule-Base Partitioning Approach

An alternate approach to partitioning is the rule-partitioning approach, in which we partition the rule-set so that the above mentioned goals are obtained. To do this, we create a *rule-dependency graph*. In a rule-dependency graph, each rule is represented by a vertex and each edge indicates that a clause in the head of a rule (r1) is present in the body of the rule (r2). This means that a tuple generated due to r1 will be used by r2. Thus we need to minimize the number of such edges that are cut, because a cut will mean that a tuple has to be communicated. To improve the partitioning, we can weigh the edges of the graph based on the number of triples they may contribute. For example, let us assume that r1 is dependent on r2 and r3 on r4. If r1 produces many more triples than r3 then the edge between r1 and r2 should be weighed more than the edge between r3 and r4. When available, a priori knowledge about the distribution of different predicates in the dataset can be used to weigh the edges of the graph. The graph obtained is then partitioned using the standard graph partitioning algorithm. The algorithm is summarized in algorithm 2:

Once a rule-base partitioning is produced, each partition (sub-set of the rules) is used in a node and applied to the

Algorithm 2 Rule partitioning

Input: Rule-base created from an ontology

Output: Partition of the rule-base

- 1: Create rule dependency graph as follows;
 - 2: Each rule is represented by a vertex
 - 3: If the head of a rule contains a clause that is in the body of a rule then add an edge between the two rules
 - 4: Partition the rule-dep graph to minimize edge cut, balance no. of rules in each partition (standard graph partitioning)
 - 5: **return**
-

original set of tuples. When a new tuple is generated, the tuple is matched with the sub-goals in the body of the rules in the other partitions to determine if it can be potentially used in it. It is then sent to each of the partitions for which it is a potential match.

IV. PARALLEL ALGORITHM

The generic algorithm for parallel processing of OWL data is shown in algorithm 3. The algorithm is very similar to the parallel algorithm for datalog programs, presented in [11], [12], [13]. The input to the parallel reasoner is the set of base tuples and a rule-base that is compiled from an OWL ontology. The master node, partitions either the data-set or the rule-base and sends the appropriate partition to each processor in the system. In the case of data partitioning approach, the base-tuples received at each processor, is a sub-set of the input tuples and the rule-base is the same as the original. On the other hand in the rule-base partitioning approach, the base tuples assigned to each partition is the same as the original tuples presented to the algorithm whereas the rule-base is a subset of the original rule-base. Apart from this, the master node also sends a partition table to each processor.

The algorithm at each node, works in *rounds*; in each round, every processor in the system, applies the rule-base to the data set to obtain the inferred tuples. It then checks if any of the inferred tuples are to be transmitted to another processor. The exact method for determining if a tuple should be sent to another partition depends on the partitioning strategy. For the data partitioning scheme, the partition/owner table which contains the information regarding which processor in the system owns a node is used to determine if a tuple needs to be transmitted to another node(s). For the rule-based partitioning, we match the newly generated with all the rules of other partitions to determine if it can trigger any of them. The tuple is sent to all the system, in which it can be used. Once the messages from all the processors are received, the next round ensues, with the additional tuples are added to the tuples produced at the end of the previous rounds used as the input tuples and the process is repeated. 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. Note that the master node itself has no role to play once the initial partition is done and input supplied to each node. It can

therefore, be used as one of the nodes of the system and thus be used to process a partition.

Algorithm 3 Parallel Reasoning

Input: Initial base tuples, rule-base

Output: Base tuples and Inferred tuples

- 1: Partition the data or rule-base. Assign a partition to each node in the system.
At each node:
 - 2: **while** !terminate **do**
 - 3: Create all the new tuples for the given rule base and initial base-tuples
 - 4: Send any of the newly generated tuples to other processors as necessary
 - 5: Receive tuples from other processors add them to the base tuples.
 - 6: **end while**
 - 7: **return**
-

One nice characteristic of this parallelization algorithm is that it uses an existing reasoner for creating additional tuples. Thus it can be built as a wrapper over an existing reasoner.

V. IMPLEMENTATION

We have used the Jena package [3] for OWL processing. 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 SLD 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. 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.

Since Jena is a Java based package, we have also developed our parallel implementation using the same language. The inter-partition communication is through the use of a shared file system. We have used shared files in our implementation mainly because, we could not find a MPI package that works with the version of java we have used. Moreover, we did not find a (free) OWL reasoner written in a more efficient language like C/C++. We however feel that this does not weaken our work because the main goal of this work is to study the effect of partitioning methodologies for OWL data sets and not to present an ideal implementation for the same. Further, we have addressed this concern by presenting the I/O overhead involved in our implementation and discussing the effect of using a more efficient communication mechanism like MPI.

VI. EXPERIMENTAL RESULTS

To measure the performance of our algorithm, we conducted experiments on a cluster of machines managed by the HPCC at USC. The experiments were conducted on AMD Opteron 2.6 GHz, 64-bit dual core machines, with main memories between 4-64GB. All these machines ran the Linux OS and Java v1.5. Each partition was executed on a separate processor core. We have used two standard benchmarks the LUBM-10 (1M triples) and UOBM-4 data-sets and our own data-set called MDC, for validating our techniques.

A. Data Partitioning

Figure 1 shows the speedups we have obtained using the data partitioning approach, implemented using the graph partitioning algorithm. For the LUBM-10 and the MDC data-sets we see super linear speedups. On the other hand for the UOBM data-set, we observe sub-linear speedups.

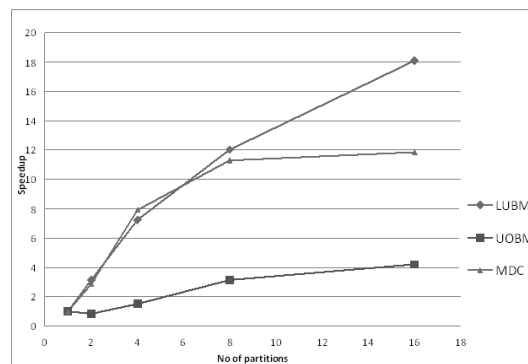


Fig. 1. Speedup for the LUBM-10, UOBM benchmarks on different number of processors.

The super-linear speedups that are observed for some benchmarks can be explained by examining the reasoning process. The backward reasoner implementation in Jena, a generic Prolog interpreter, uses SLD resolution for query answering. To materialize the KB, queries of the form find all statements with a given resource as subject is issued for each resource in the graph. In answering this query, the reasoner creates *kn* triples, where each triple has the given resource as subject and each of the *n* triples as the object. It then tries to prove that the KB entails such a triple. The worst-case complexity of this algorithm is polynomial in the number of resources in the KB. For some data-sets like LUBM and MDC, the reasoner, exhibits the worst case polynomial complexity (verified in the following sub-section). For such data-sets, our partitioning algorithm, reduces the search space for proving the theorems and thus shows super-linear speedups. However, for some data/rule-sets like UOBM, the reasoner does not exhibit worst-case complexity and scales linearly with the size of the data-set. For such data-sets we observe sub-linear speedups.

B. I/O overhead and ideal speed-up

Figure 2 shows the average amount of time spent for reasoning, IO, synchronization (waiting for other partitions to finish current round) and aggregation by the parallel algorithm for the LUBM-10 benchmark. The figure shows the maximum values over the partitions. We see that as the number of partition increases, the amount of time spent in inter-process communication (file IO in our case) and synchronization correspondingly increases. This might be of concern for a larger scale parallel reasoner, and can be improved by:

- The inter-process communication time (IO) can be reduced by using a more efficient communication mechanism like MPI.
- In our current implementation, after a partition has finished its round, it has to wait for all the partitions to finish their current rounds before proceeding to the next round. By making a partition to not wait till all other partitions to finish, but rather start immediately using all the currently received tuples will reduce the synchronization time of the algorithm.

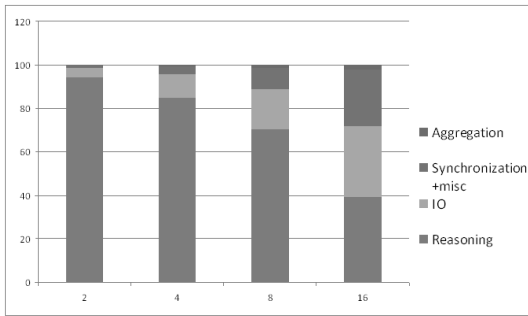


Fig. 2. Overhead of various sub-tasks of parallel processing for LUBM-10.

In figure 3 we compare the speedup achieved by our implementation with the theoretical maximum speedup for the LUBM data set. The theoretical maximum was calculated by using an empirically derived performance model for the reasoning for LUBM data set. We ran the serial reasoner on various LUBM data sets (LUBM-1, LUBM-5, LUBM-10 etc) and based on the reasoning times obtained for them, we regressed a cubic model for the execution time, as shown in figure 4. Since the worst case of the reasoning for the rule set is cubic, fitting a cubic model is reasonable. The theoretical maximum is calculated as the speedup obtained by a partition in which all the partitions are of equal size and there is no replication among the partitions. The graph shows the reasoning for the slowest partition as well as the overall reasoning time for the data-set. We see that the speedups observed are quite close to the theoretical maximum predicted by the model. Thus a more efficient communication and synchronization scheme can be further used to show much better performance, especially as the number of processors in the system is increased.

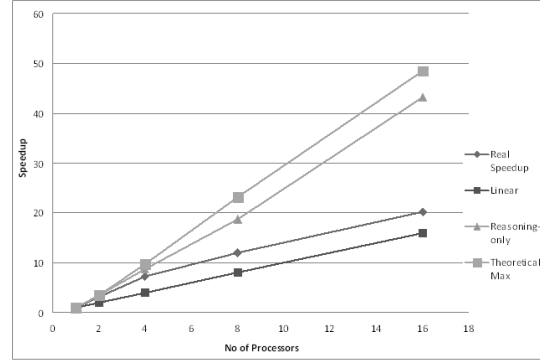


Fig. 3. Speedup for the LUBM-10 benchmark on different number of processors.

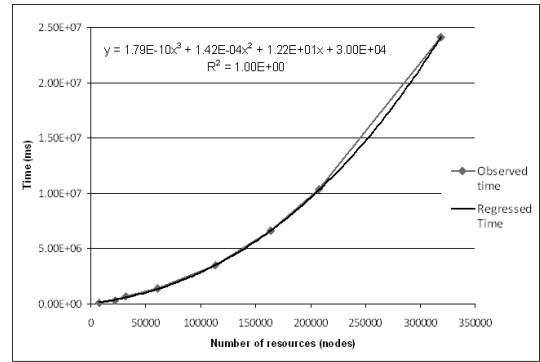


Fig. 4. Regressing a performance model from observed reasoning times for LUBM data-sets.

C. Comparison of data partitioning algorithms

Figure 5 shows the comparison between speedups obtained from the three data partitioning algorithms presented earlier¹. Table I summarizes the metrics proposed in section III for the partitioning algorithms, can be used to explain the results as well as validate the usefulness of the metrics themselves.

The domain specific partitioning performs nearly as well as the graph partitioning algorithm. Both these algorithms, create partitions that are of equal size, and have few edge-cuts across partitions. This is indicated by the fact that *bal* and the *IR* is relatively small. Note that although domain specific algorithm has the highest *bal*, the number is quite small with respect to the total number of nodes in the graph(330K). The naive hash algorithm on the other hand, performs very badly because, it does not minimize edge-cut. Therefore, the amount of duplication, i.e., the number of partitions in which the same tuple is created, is very high. E.g., for the LUBM data-set using 4 partitions, the graph based partitioning algorithm the duplication (*IR*) is nearly 10% whereas for the hash algorithm it is about 100%. The running times of all the algorithms are

¹Speedups for hash partitioning using 8 and 16 nodes not shown because, the experiments did not complete due to memory size limitations.

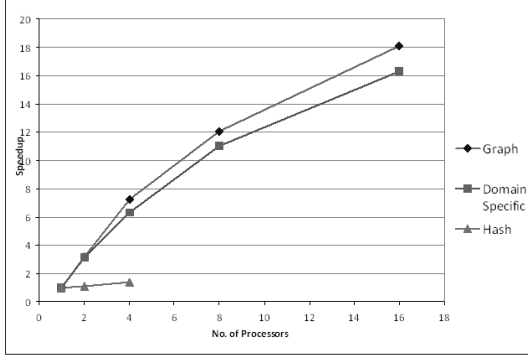


Fig. 5. Comparison of performance of the two data-partitioning algorithms for LUBM-10.

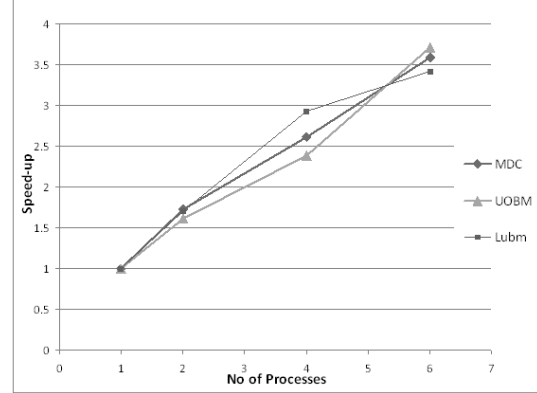


Fig. 6. Speedup for the different benchmarks for rule-base partitioning.

three orders magnitude smaller than the inferencing time and thus negligible.

No of partitions	Algorithm	Bal	OR	IR	Part. Time(s)
2	Graph	1500	0.1	0.07	240
	Dom sp.	9186	0.1	0.07	156
	Hash	168	0.5	0.7	120
4	Graph	1533	0.1	0.13	235
	Dom sp.	10950	0.1	0.11	154
	Hash	772	0.95	1.5	120
8	Graph	986	0.15	0.16	235
	Dom sp.	9304	0.13	0.13	154
	Hash	788	-	2.1	120
16	Graph	604	X	0.19	235
	Dom sp.	2963	X	0.19	154
	Hash	2800	-	1.3	120

TABLE I
PARTITIONING METRICS FOR THE LUBM DATA-SET

D. Rule-base Partitioning

Figure 6 shows the speedups obtained by the rule-partitioning approach. We show the results for the three benchmarks: LUBM, UOBM, and MDC. We had to modify the implementation for the rule-base partitioning to use shared memory to communicate rather than files, because the volumes of data being communicated across processors was much higher. The results show sub-linear but monotonic speedups. Since all of these rule-sets are fairly small, we have only conducted experiments on a small number of processors.

VII. RELATED WORK

To the best of our knowledge ours is the first work to examine the problem of parallelizing OWL inferencing. However, much work has been done in the closely related areas of parallelizing rule based and deductive database systems. We present a brief overview of the literature of such systems below.

The existing work can be classified based on the kind of applications of the rule based system for which the parallelization technique was devised. Techniques developed for

production systems (OPS5 is a language that is commonly used) are presented in [14], [15]. In this class of applications, unlike our application domain, the data sizes are generally small where as the rule-bases could be quite large. Hence the techniques devised in these works are not ideally suited for our problem. Techniques devised for *deductive databases*, which use the Datalog language for rules have been presented in [11], [12], [13]. This is the body of work that is most relevant to us, as OWL reasoning can be implemented using a datalog reasoner and also because like in deductive databases, the data-sets are much larger than rule-bases. The parallel algorithm that we have presented is very similar to those proposed in the above works. However, to the best of our knowledge all the work in this community is theoretical and we are not aware of a system that actually implements the above techniques for a Datalog implementation. Moreover there are no studies based on standard benchmark data-sets to help us compare our results with the work done in this community or map them directly to the OWL reasoning problem.

The data partitioning based approach to parallelizing deductive data-bases have been proposed [12], [13], [16]. In [12], [13], the authors propose an abstract hash function, which maps a tuple of data on to a processor, for partitioning the data. The main challenge of such a system is to choose an appropriate hash function for the data set to partition the computational workload equally among the processors and minimize communication of tuples across partitions. Further, the authors propose some useful guidelines for such a hash function in [12]. The partitioning algorithms that we propose can be seen as realizations of such abstract hash functions for datalog programs realizing OWL semantics and employ some of the learnings from the above mentioned works. Two other data partitioning approaches are proposed in [16]. In the cluster based partitioning a distance metric of a tuple referenced by a rule is used to partition the data-set by using a clustering algorithm. This is somewhat similar to the graph partitioning approach that we have used. In statistics based approach, the assumption is that of a (more or less)

stationary dataset. Thus statistics could be gathered and used to perform partitioning. Such a policy is mentioned in [16]. Our domain specific partitioning algorithm is similar to this approach because it also uses the characteristics of a particular data-set to create good partitions.

The rule-base partitioning approach for datalog programs has been presented in [17]. Our own rule-base partitioning approach is somewhat similar to theirs although our algorithm is specific to OWL datalog programs. Two other partitioning approaches that we plan to explore in our future work are the hybrid partitioning approach [18] and the control partitioning approach [19]. In hybrid partitioning both the rule-set as well as data-set are partitioned to obtain better results. In the control partitioning typically seen in Prolog systems, different branches or flows of control are explored in parallel.

The existing systems can also be classified based on how the systems handle the issue of load balancing to provide equitable workloads to each node in the system. In static load balanced systems [11], the workload partitioning is done at the beginning and is not modified as data is processed by the nodes. This is the approach we have used in our system. On the other hand, in dynamic load-balancing system like [13], [20] reallocates workloads, if the initial partitioning scheme did not provide an balanced partition. Our own system uses a static load balancing strategy, which as we see based on our results, works quite well.

VIII. CONCLUSIONS

We have demonstrated two techniques for partitioning the workload of the OWL reasoning process and hence parallelize it. Our approach can be used not only on parallel clusters, but also on the popular multi-core processors. Our data 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 simple technique to partition the datasets and to process them in parallel. The results we have obtained on standard benchmarks are promising for the data-partitioning approach with speedups up to 18x being observed on a 16 node parallel cluster for a particular benchmark. In rule partitioning approach, the rule-base is partitioned and the simpler rule-base is applied to the complete data-set. For data-sets which are dense graphs, the rule partitioning approach provides better speedups than the data partitioning approach. Moreover the rule-partitioning approach is much cheaper than the data partitioning approach because, the data-sets are typically much larger than the rule-sets. Thus it might be useful for dynamic load balancing situations, where the data-set is initially partitioned and during later rounds rule-sets are partitioned for load balancing.

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] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
- [2] B. N. Groszof, I. Horrocks, R. Volz, and S. Decker. "Description logic programs: combining logic programs with description logic," in *WWW*, 2003, pp. 48–57.
- [3] "Jena semantic web framework, <http://jena.sourceforge.net/>."
- [4] "Owlrim semantic repository, <http://www.ontotext.com/owlrim>."
- [5] "Oracle semantic technologies, http://www.oracle.com/technology/tech/semantic_technologies."
- [6] H. J. ter Horst, "Combining rdf and part of owl with rules: Semantics, decidability, complexity," in *International Semantic Web Conference*, 2005, pp. 668–684.
- [7] V. Vianu, "Rule-based languages," *Annals of Mathematics and Artificial Intelligence*, vol. 19, no. 1-2, pp. 215–259, 1997.
- [8] R. Soma and V. K. Prasanna, "A data partitioning approach for parallelizing rule based inferencing for materialized owl knowledge bases (manuscript)." [Online]. Available: <http://www.scf.usc.edu/rsoma/papers/data-partitioning.pdf>
- [9] U. Elsner, "Graph partitioning - a survey," 1997.
- [10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
- [11] S. Ganguly, A. Silberschatz, and S. Tsur, "A framework for the parallel processing of datalog queries," in *SIGMOD Conference*, 1990, pp. 143–152.
- [12] 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, 1995.
- [13] O. Wolfson and A. Ozeri, "Parallel and distributed processing of rules by data reduction," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 3, pp. 523–530, 1993.
- [14] A. Gupta, C. Forgy, A. Newell, and R. G. Wedig, "Parallel algorithms and architectures for rule-based systems," in *ISCA*, 1986, pp. 28–37.
- [15] J. Amaral, "A parallel architecture for serializable production systems," Ph.D. dissertation, The University of Texas at Austin, Austin, TX, 1994.
- [16] S. Stolfo, H. Dewan, D. Ohsie, and M. Hernandez, "A parallel and distributed environment for database rule processing: open problems and future directions," *Emerging Trends in Database and Knowledge-Based Machine*, 1995.
- [17] D. A. Bell, J. Shao, and M. E. C. Hull, "A pipelined strategy for processing recursive queries in parallel," *Data Knowl. Eng.*, vol. 6, pp. 367–391, 1991.
- [18] J. Shao, D. A. Bell, and M. E. C. Hull, "Combining rule decomposition and data partitioning in parallel datalog program processing," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991)*, Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991. IEEE Computer Society, 1991, pp. 106–115.
- [19] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey," *Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, 2001.
- [20] H. M. Dewan, S. J. Stolfo, M. Hernández, and J.-J. Hwang, "Predictive dynamic load balancing of parallel and distributed rule and query processing," in *ACM-SIGMOD Intl. Conf. on Management of Data*, 1994, pp. 277–288.