Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples

Jesse Weaver and James A. Hendler

Tetherless World Constellation, Rensselaer Polytechnic Institute, Troy, NY, USA
{weavej3,hendler}@cs.rpi.edu

Abstract. In this paper, we consider the problem of materializing the complete finite RDFS closure in a scalable manner; this includes those parts of the RDFS closure that are often ignored such as literal generalization and container membership properties. We point out characteristics of RDFS that allow us to derive an embarrassingly parallel algorithm for producing said closure, and we evaluate our C/MPI implementation of the algorithm on a cluster with 128 cores using different-size subsets of the LUBM 10,000-university data set. We show that the time to produce inferences scales linearly with the number of processes, evaluating this behavior on up to hundreds of millions of triples. We also show the number of inferences produced for different subsets of LUBM10k. To the best of our knowledge, our work is the first to provide RDFS inferencing on such large data sets in such low times. Finally, we discuss future work in terms of promising applications of this approach including OWL2RL rules, MapReduce implementations, and massive scaling on supercomputers.

1 Introduction

At present, the semantic web consists of ever-increasing Resource Description Framework[1] (RDF) data. In RDF, the fundamental unit of information is a triple; a triple describes a relationship between two things. Some triples in conjunction with each other can give rise to new knowledge. Consider rule *rdfs2* from [1]:

$$(?a \text{ rdfs:domain } ?x) \wedge (?u \ ?a \ ?y) \rightarrow (?u \text{ rdf:type } ?x)$$

Deriving such inferences in the semantic web poses several challenges. First, data on the web is distributed making it difficult to ensure that the appropriate triples (e.g., rdfs:domain) are discovered together to derive the appropriate inferences (e.g., rdf:type). Second, the semantic web continues to grow creating vast amounts of information. Computation capable of scaling to large data sets is necessary. Third, the amount of time required to derive inferences should be reasonable (which depends on a use case).

[1] http://www.w3.org/RDF/

These three issues of dependency, scalability, and time must be addressed to make inferencing on the semantic web practical and useful. We look at how to resolve these issues for computing the finite RDF Schema (RDFS) closure of large data sets. We find that the RDFS rules have certain properties which allow us to solve the issue of dependency rather easily. Scalability and time are addressed by forming an embarrassingly parallel[2] algorithm for such rules that allows for strong scaling.

We consider the problem of materializing the complete finite RDFS closure in a scalable manner. We define the finite RDFS closure as follows.

Definition 1. *We define the* ***finite axiomatic triples****—denoted T_{rdfs}—as the RDF and RDFS axiomatic triples [1] minus any triples that describe resources of the form rdf:_#.*

Definition 2. *We define the* ***finite RDFS rules*** *as the set of rules concerning literal generalization (lg, gl) and the RDF and RDFS entailment rules (rdf1-2, rdfs1-13)[1], and also the following rule which we call cmp:*

$$\textit{(?n rdf:type rdfs:Resource)} \wedge \textit{?n is of the form rdf:\_\#} \longrightarrow \textit{(?n rdf:type rdfs:ContainerMembershipProperty)}$$

Definition 3. *The* ***finite RDFS closure*** *is defined the same way as the RDFS closure [1] with the following exceptions: (1) the finite axiomatic triples are used instead of all of the RDF and RDFS axiomatic triples; and (2) rule cmp is included in the last step of rule applications.*

To our knowledge, only a few systems exist that produce the complete RDFS closure, none of which scale to large data sets. We address this problem by defining a partitioning scheme and showing that the finite RDFS rules can be applied to such a partitioning to produce the finite RDFS closure. In addition to deriving an embarrassingly parallel algorithm, we discuss other challenges such as parallel file I/O for RDF data and handling blank nodes. We present and evaluate an implementation of the algorithm written in C using the Message Passing Interface[3] (MPI) on a cluster of large memory Opteron machines for large subsets of the LUBM 10,000-university data set, scaling up to 128 processes. We also give an evaluation for the amount of duplicate inferences in the output, and promising applications are discussed as future work.

2 Related Work

The work most related to ours is MaRVIN [3, 4] and the parallel Web Ontology Language[4] (OWL) inferencing work by [5, 6].

[2] An embarrassingly parallel computation is “the ‘ideal’ computation from a parallel computing standpoint—a computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor.” [2]

[3] http://www.mpi-forum.org/

[4] http://www.w3.org/2004/OWL/

MaRVIN provides sound, anytime, and eventually complete RDFS reasoning using a "divide-conquer-swap" strategy. Every process uses a reasoner and processes a fraction of the data to locally produce any possible inferences. A scoring mechanism is used to determine which triples are most useful for further inferencing, and these triples are exchanged between processes in an attempt to mix up the triples and find more inferences. Their evaluation in [4] shows presumably nearly complete reasoning on 14.9 million SwetoDBLP[5] triples in roughly 23 minutes.

[6] presents an approach to rule-based reasoning for OWL ontologies with so-called "OWL Horst" (perhaps better known as pD*) semantics [7]. Before parallel inferencing occurs, a fair amount of preprocessing is required. OWL ontologies are compiled into rules and a partitioning is determined ahead of time. Four partitioning approaches are presented, three of which are classified as data partitioning and one of which is classified as rule partitioning. Data partitioning gives each process a fraction of the data and all of the rules, while in rule partitioning, each process receives all of the data and a fraction of the rules. While it is uncertain what execution times were actually achieved, they indicate that they were able to achieve a speedup of 18 for 16 processes on the Lehigh University Benchmark[6] (LUBM) [8] 10-university data set (approximately one million triples). Other data sets evaluated, however, did not see as much speedup.

These two projects have not shown scaling to the extent that we demonstrate for our system, and they also handle only limited subsets of existing reasoning standards or perform approximation. We show scaling that has not been previously demonstrated in other systems (to the best of our knowledge) for complete finite RDFS reasoning. Furthermore, our inferencing times are smaller than those reported for the two previously mentioned works.

Also, BigOWLIM[7] reports reasoning times on large data sets, but it is uncertain how to compare the performance of BigOWLIM with our system. BigOWLIM is a semantic web repository that reportedly can perform some level of reasoning on 8 billion statements. Loading, inferencing, and query evaluation of the LUBM benchmark on LUBM8k took 15.2 hours. Our work focuses on inferencing as a large computation without concerns of storage or indexing, so the two systems are not quite comparable.

Less related works include reasoning over distributed hash tables [9, 10]; potentially parallel evolutionary query answering [11]; composition of approximate, anytime reasoning algorithms executed in parallel mentioned at the end of [12]; proposals of parallel computation techniques for ontology reasoning [13]; and approaches for exploiting vertical parallelism in tableaux-based description logic reasoning [14] .

The SOAR work [15–17] uses assumptions and observations similar to some of those used in this paper. While SOAR focuses on a scalable, disk-based, reasoning system using some RDFS and OWL-based rules on a single machine,

[5] http://lsdis.cs.uga.edu/projects/semdis/swetodblp/
[6] http://swat.cse.lehigh.edu/projects/lubm/
[7] http://ontotext.com/owlim/benchmarking/lubm.html

our system differs greatly from SOAR in that it is not disk-based, it is parallel, and it provides complete (finite) RDFS reasoning.

3 Our Approach to Parallel RDFS Reasoning

Our approach takes advantage of modern parallel computation techniques to compute the finite RDFS closure of large data sets. Previous work has used approximation to achieve higher scalability while other work focuses on minimizing dependencies in partitioning the work load. The former has the disadvantage of sacrificing soundness and/or completeness while the latter often requires time-intensive sequential computation to prepare the data for parallel reasoning. We focus on finding properties of RDFS reasoning that allow for natural parallelization in all parts of the computation. Therefore, we maintain soundness and completeness without requiring any cumbersome preparation of the data. We discuss challenges and their solutions in the following subsections.

3.1 Workload Partitioning

Workload partitioning involves breaking down the problem in such a way that it can be executed in parallel. Ideally, each process should take the same amount of time for the entire computation. In [6], the distinction between data partitioning and rule partitioning is made. For RDFS reasoning, we consider it self-evident that data partitioning is the appropriate approach considering that RDFS reasoning has fewer than 20 rules while data sets could scale into billions of triples and beyond.

Before continuing, we define a few terms used throughout this paper:

Definition 4. *An **ontological triple** is a triple used in describing an ontology and from which significant inferences can be derived. For RDFS, these are triples with predicate rdfs:domain, rdfs:range, rdfs:subClassOf, or rdfs:subPropertyOf and also triples with predicate rdf:type with object rdfs:Datatype, rdfs:Class, or rdfs:ContainerMembershipProperty.*

For clarification, we emphasize that not all triples with predicate *rdf:type* are ontological triples in the context of RDFS. Only *rdf:type*-triples for which the object is *rdfs:Datatype*, *rdfs:Class*, or *rdfs:ContainerMembershipProperty* are considered ontological.

Definition 5. *An **assertional triple** is any triple that is not an ontological triple.*

Considering the finite RDFS rules, a particular property becomes apparent. We find that each rule has at most one triple pattern in its body that can match an assertional triple. For example, consider rule *rdfs3*:

$$(?a \text{ rdfs:range } ?x) \wedge (?u\ ?a\ ?v) \rightarrow (?v \text{ rdf:type } ?x)$$

The first triple pattern of the rule body will not match assertional triples but only ontological triples. The second triple pattern, however, could match any triple.

Definition 6. *An M1 rule is a rule whose body has at most one triple pattern which can match an assertional triple.*

Proposition 1. *By inspection, the finite RDFS rules are all M1.*

Proposition 1 gives way to our partitioning scheme. By allowing each process to have all ontological triples but only a fraction of the assertional triples, we can distribute the data in such a way that the workload can be executed in parallel. We consider this approach to be reasonable since ontologies tend to be fixed data sets that are relatively very small compared to the potentially ever-increasing assertional data.

Definition 7. *An* ***abox partitioning*** *is a data partitioning scheme in which each process/partition gets all ontological triples and a fraction of the assertional triples.*

Theorem 1. *A single application of M1 rules to the partitions in an abox partitioning produces the same inferences as in a single partition with all triples. More formally, define a single application of a rule r to a set of triples G, denoted $r(G)$, as the triples resulting from satisfying the antecedent of r without adding such triples back into G. Then, given an M1 rule r_m and an abox partitioning $(G_1, G_2, \ldots, G_n)$ of a set of triples G, $r_m(G) = \bigcup_{i=1}^{n} r_m(G_i)$.*

Proof. Since an M1 rule needs at most one assertional triple to be satisfied, it can be satisfied on the partition that has such a triple in the same way it is satisfied on a single partition with all triples since all partitions have all ontological triples. $\square$

Therefore, our approach is to give each process a partition of an abox partitioning, and that process will apply all finite RDFS rules until no more inferences can be found. This is shown in Algorithm 1. Line 1 is to be interpreted as parallelism in which i denotes the rank of the process. Line 4 iterates over the finite RDFS rules in the appropriate order to satisfy data dependencies between consequents and antecedents of rules such that only a single pass over the rules is required.

It remains to be shown that this algorithm correctly produces the RDFS closure. Although M1 rules produce all the appropriate inferences when applied once, it must be shown that placing the inferences in the partition from which they were derived sufficiently maintains the abox partitioning so that subsequent applications of the rules will also produce the correct inferences.

Lemma 1. *If an M1 rule sufficiently maintains an abox partitioning after adding its inferences back to the partition from which they were derived, then the M1 rule can be applied multiple times and produce sound and complete inferences.*

Algorithm 1: Parallel RDFS Inferencing Algorithm

```
Input: A set of assertional triples $T_A$, a set of ontological triples $T_O$, and a
       number of processes $p$ . $T_{rdfs}$ is the set of finite axiomatic triples.
Output: All triples together with all inferences from the computation of finite
        RDFS closure.
// outer loop denotes parallelism where $i$ is rank of process
1  for $i = 0$ to $p - 1$ do
2      $T_{A_i} = \{ t \mid t \in T_A \land t \notin T_{A_j}, \forall j \neq i \}$
       // $T_{A_i}$ contains roughly $|T_A|/p$ triples from $T_A$
       // that are given only to process $i$
3      $T_i = T_{A_i} \cup T_O \cup T_{rdfs}$
4      foreach rule $\in$ finite RDFS rules do
5          repeat
6              apply rule to $T_i$ to get inferences
7              add inferences to $T_i$
8          until no new inferences
9      end
10 end
11 return $\bigcup_{i=0}^{p-1} T_i$
```

Proof. The proof is intuitive. Since M1 rules produce sound and complete inferences in a single application to an abox partitioning, if the result after adding the inferences is an abox partitioning, the M1 rules can be applied again to produce sound and complete inferences. □

Definition 8. *Say that a rule that fits the description in Lemma 1 is* ***abox partitioning safe (APS)****.*

Proving that a rule (or set of rules) is APS consists of proving it is M1 and that it sufficiently maintains the abox partitioning. (We return to the issue of sufficiency later in the paper.) We define several classes of M1 rules that sufficiently maintain an abox partitioning, and thus, such rules are APS. Before doing so, we make the following assumption.

Axiom 1. *Other than those mentioned in the axiomatic triples or those produced from the entailment rules, the resources in the RDF and RDFS vocabulary have no superclasses, subclasses, superproperties, subproperties, domains, or ranges.*

This axiom allows us to disregard odd cases that might occur if such triples were included (like if rdfs:Class has a superclass). Such triples could modify the semantics of RDFS, and so we simply disallow them. We now go on to prove that the finite RDFS rules are APS using sketch proofs to be concise and excluding proofs of propositions that are straightforward.

Definition 9. *Say that a rule is* ***abox partitioning easy (APE)*** *if it is M1 and produces only assertional triples.*

Theorem 2. *APE rules are APS.*

Proof. Adding the produced assertional triples from APE rules to any partition results in an abox partitioning since the ontological triples are unaffected, so all partitions still have all ontological triples. □

Proposition 2. *By inspection, rules lg, gl, rdf1, rdf2, rdfs1, rdfs4a, rdfs4b, and rdfs7 are APE.*

Definition 10. *Say that a rule is* ***abox partitioning ontological (APO)*** *if the triple patterns in its body can match only ontological triples. (Note that this implies they are M1.)*

Theorem 3. *APO rules are APS.*

Proof. Since all partitions have all ontological triples, such rules will produce all of their inferences on all partitions. If an APO rule produces ontological triples, they will be produced on all partitions, and thus all partitions still have all ontological triples. □

Proposition 3. *By inspection, rules rdfs5, rdfs8, rdfs10, rdfs11, rdfs12, and rdfs13 are APO.*

Definition 11. *Say that a rule is* ***abox partitioning friendly (APF)*** *if it is M1 and it produces ontological triples only when the body is satisfied by only ontological and/or axiomatic triples.*

Theorem 4. *APF rules are APS.*

Proof. Since all partitions have all ontological triples and axiomatic triples (every partition adds the axiomatic triples at the beginning), when APF rules produce ontological triples, they are produced on all partitions, and thus, all partitions still have all ontological triples. □

Proposition 4. *By inspection, rules rdfs2, rdfs3, and rdfs9 are APF.*

Definition 12. *Say that a rule is* ***abox partitioning trivial (APT)*** *if it is M1 and the ontological triples it produces do not contribute to the inferencing of new triples that would not otherwise be inferred by other rules.*

Theorem 5. *APT rules are APS.*

Proof. This is where "**sufficiently** maintains an abox partitioning" from Lemma 1 is important. APT rules do not necessarily ensure that all partitions will have all ontological triples. Instead, they ensure that the ontological triples that they produce are insignificant for further inferencing. Therefore, even though such triples may have the form of ontological triples, they fail to meet the part of the definition of ontological triple which states that significant inferences are derived from them. Therefore, we can disregard the ontological triples produced by APT rules, including them only for completeness. □

Proposition 5. *Rule rdfs6 is APT.*

Proof. We include a brief proof for this proposition since it is a little less intuitive than the other propositions. *rdfs6* produces triples of the form (?u rdfs:subPropertyOf ?u). Such triples can help to satisfy rules *rdf1*, *rdfs2*, *rdfs3*, *rdfs4a*, *rdfs4b*, *rdfs5*, and *rdfs7*. The first five rules are intuitive and are not elaborated upon here. Using triples produced by *rdfs6*, rules *rdfs5* and *rdfs7* merely produce triples that already exist. □

Definition 13. *Say that a rule is* ***abox partitioning dynamic (APD)*** *if it is M1 and it produces parts of the ontology (ontological triples) only if the partition needs them to produce inferences that otherwise would not be produced.*

Theorem 6. *APD rules are APS.*

Proof. The concern is that APD rules may create some ontological triples on some partitions and not on others. By definition, though, if the other partitions need these ontological triples to produce sound and complete inferences, then they would have been produced on that partition also by the APD rules. Therefore, all partitions can be considered to have all ontological triples produced by APD rules in the sense that all partitions have the APD rules and would produce the triples if needed. □

Proposition 6. *Rule cmp is APD.*

Proof. If a *rdf:_#* resource is mentioned in the triples of a partition, then the triple (*rdf:_#* rdf:type rdfs:Resource) will eventually be produced. (If used as a property, by way of *rdf1* and *rdfs4a*; if used as a subject, by way of *rdfs4a*; and if used as an object, by way of *rdfs4b*.) Then, *cmp* is satisfied by that triple, and the appropriate ontological triples for *rdf:_#* are produced. □

Corollary 1. *By Theorems 1-6 and Propositions 1-6, all of the finite RDFS rules are APS. Therefore, the finite RDFS closure can be computed in parallel using abox partitioning.*

3.2 Parallel File I/O for RDF Data

As mentioned, we desire that our approach require no preprocessing of the data; everything should be performed in parallel. Therefore, we must determine a way to read and write RDF data in parallel. Most parallel I/O approaches use a "chunking" method in which each process gets a fairly even number of bytes from the data file. However, in RDF, the fundamental unit of data is the triple, not the byte. RDF syntaxes are string-based and therefore cannot be divided into mere bytes assuring that each process will get a set of *complete* triples. Some RDF syntaxes such as RDF/XML [18] make this particularly difficult. If an RDF/XML file is divided into portions of bytes, it becomes extremely difficult to determine which triples are complete and which triples are incomplete in that portion. We take advantage of the simple RDF syntax N-triples[8].

In N-triples syntax, every triple occupies a single line. When a process reads its portion of bytes, it can determine where the first complete triple begins by locating the first end-line character, and it can determine where the last complete triple ends by locating the last end-line character. Each process i sends the triple fragment at the beginning of its bytes to process $i-1$ (process 0 exempted), and each process uses this fragment to complete the last (partial) triple. In this way, each process has a set of complete triples, and we assume that reading a fairly equal number of bytes will generally result in a fairly even number of triples.

This method of reading triple sets from an N-triples file on disk is used to partition the assertional triples as described in line 2 of Algorithm 1. Thus, we require no preprocessing of data, although we do require that it is in N-triples format.

3.3 Distributed Blank Nodes

After partitioning the assertional triples, each process essentially has its own RDF graph. Thus, some meaning is lost if blank nodes are distributed. In the original graph, we know that two blank nodes are the same because they have the same label within the same graph. Now, however, we have partitioned the graph into smaller graphs, and there is no guarantee that two blank nodes with the same label in different graphs are actually the same node. (Note that since we require data in N-triples format, all blank nodes have labels.) To resolve this problem, each blank node is replaced by a special URI with scheme "b" and with URI body equal to the blank node identifier. This is done while reading the data to ensure that we will always be able to refer to blank nodes uniquely. For example, blank node _:bnode123 would become <b:bnode123>. Then, as the results are written to file, the special URIs are turned back into blank nodes.

This handles the case of already-existing blank nodes, but it becomes more difficult when applying rules *lg*, *rdf2*, and *rdfs1* in which blank nodes are uniquely allocated to literals. We use a similar approach as with the already-existing blank nodes. We turn the literals into URIs by encoding the literal into an appropriate blank node identifier and then using that identifier as the URI body of a special URI with scheme "l." We use a simple encoding that we call a z-encoding.

Definition 14. *A **z-encoding** of a literal is generated as follows. The literal is first represented in N-triples syntax (including unicode escapes). Then, each character in the string that is not 0-9, A-Z, or a-y is replaced by "zHH" where HH is the hexadecimal representation of the character.*

Using the special l-scheme URIs with z-encoded labels ensures that each process produces the same blank node identifier for each blank node allocated to a literal, and the blank nodes are referred to in a consistent way across processes. These URIs are also converted into blank nodes during output.

[8] http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/#ntriples

4 Implementation

Our implementation is written in C using Redland[9] for in-memory RDF storage and query processing. We use Redland's tree storage structure for efficient loading and querying of RDF data. We implement each finite RDFS rule as a SPARQL Protocol and RDF Query Language[10] (SPARQL) query followed by a function call. The SPARQL query serves to find all potential matches for the rule and the function serves to further restrict the results if needed (e.g., the well-formed requirement in *rdf2*) and produce the appropriate triples. Most rules are implemented as a simple CONSTRUCT query followed by a function that simply adds the resulting triples. For example, *rdfs3*'s query (prefix declarations omitted) is:

CONSTRUCT { ?v rdf:type ?x } { ?a rdfs:range ?x . ?u ?a ?v }

This rule is simply executed and the results added to the process' triples. A more complicated case would be rule *lg*:

CONSTRUCT { ?u ?a ?l } { ?u ?a ?l . FILTER(isLITERAL(?l)) }

In this case, the query is used to find all triples with literal objects, but an additional function is needed to actually allocate a blank node to ?l.

We use MPI for parallel I/O and interprocess communication (only necessary when reading assertional triples). However, unlike the set-union operator on line 11 of Algorithm 1, we simply write the results of each partition/process to a file without eliminating any duplicate triples between partitions. Ensuring that only one process has any given triple (i.e., removing duplicates) would take away from the embarrassingly parallel nature of the algorithm, and so to emphasize the scalability of the algorithm, we do not remove duplicates. Writing to different files allows for better parallel I/O performance since processes will not have to compete for file locks, and all the files will be "self-describing" in the sense that they each have the ontology. The downside, however, is that the overall resulting data set will be larger than necessary, so we present an evaluation in the following section for how much duplication of inferences actually occurs.

The overall process involves initialization of the environment, loading ontological triples from an N-triples file, loading assertional triples from a different N-triples file, performing inferencing, writing results to separate files (one per process), and finalizing the environment.

5 Evaluation

For a data set, we generated the LUBM 10,000-university data set (LUBM10k) which, when the generated OWL files are translated directly to N-triples files,

[9] http://librdf.org/
[10] http://www.w3.org/TR/rdf-sparql-query/

contains 1,382,072,494 triples. We broke LUBM10k down into subsets by continually halving the data set, generating data sets that we denote as LUBM10k/2, LUBM10k/4, ..., LUBM10k/1024. LUBM10k/1024 is the smallest data set for which we perform an evaluation, and it contains 1,349,680 triples. LUBM10k/4 is the largest data set for which we perform an evaluation, and it contains 345,518,123 triples.

We perform our evaluation on the Opteron blade cluster at Rensselaer Polytechnic Institute's (RPI) Computational Center for Nanotechnology Innovations[11] (CCNI) using only the large memory machines. Each machine is an IBM LS21 blade server running RedHat Workstation 4 Update 5 with two dual-core 2.6 GHz AMD Opteron processors with gigabit ethernet and infiniband interconnects and system memory of 16 GB. We read and write files to/from the large General Parallel File System[12] (GPFS) which has a block size of 1024 KB, scatter block allocation policy, and 256 KB RAID device segment size using a RAID5 storage system.

We ran each job with an estimated time limit of 30 minutes which—due to the nature of the job queuing system—lessened the waiting time for execution. After 30 minutes, the scheduler terminates the job if it is not finished. We found this to be reasonable since in our experience, if it took longer than 30 minutes, it was usually because memory usage was at maximum capacity and the time to finish (if possible) would far exceed 30 minutes due to swapping. When the job is run, it has full control over its nodes. Four processes are on one machine since each machine has four cores, and this causes contention for memory between processes. The main source of contention, though, is among our processes and among external processes in the CCNI that may create a high demand of service on the disk, thus slowing disk I/O. We attempted to perform our evaluation at times when the CCNI was least used to try and reduce competition for disk service. Timings (wall clock, not just CPU) were measured using RDTSC (ReaD Time Stamp Counter).

5.1 Performance

Since the algorithm is embarrassingly parallel, it is no surprise to see in Figure 1 that the time to inference halves as the number of processes doubles. Similarly, as the size of the data set doubles, the time to inference doubles.[13] On 128 processes, LUBM10k/1024 takes 1.10 seconds, and LUBM10k/4 takes 291.46 seconds.

Figure 2 shows that the overall time of the computation is generally linear, but as the smallest data set is run on a larger number of processes, the speedup

[11] http://www.rpi.edu/research/ccni/

[12] http://www-03.ibm.com/systems/clusters/software/gpfs/index.html

[13] While LUBM data is well-structured and evenly distributed, data that is more uneven (high skew) would likely exhibit similar linear scalability, especially as the number of processes increases. Further performance evaluation using different data sets is part of future work.

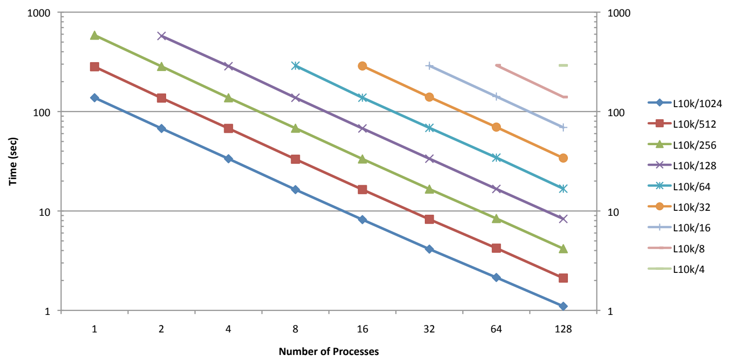

Fig. 1. Time for inferencing only, averaged across processes, for different-size data sets and varying number of processes.

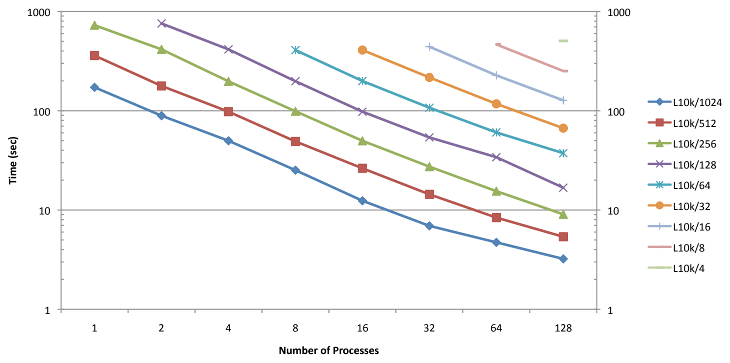

Fig. 2. Overall time averaged across processes, for different-size data sets and varying number of processes.

decreases. This behavior is likely caused by the small amount of work per process when a larger number of processes are employed. Figure 3 shows this more clearly for LUBM10k/1024 on varying numbers of processes. Time to infer tends to dominate the computation time and decreases as the number of processes increases. The time to initialize is generally very small, but increases with the number of processes. For 128 processes, it took roughly the same amount of time to infer as it did to initialize the environment. Therefore, as the amount of work per process gets smaller, the overall time reaches the overhead cost of computing in parallel.

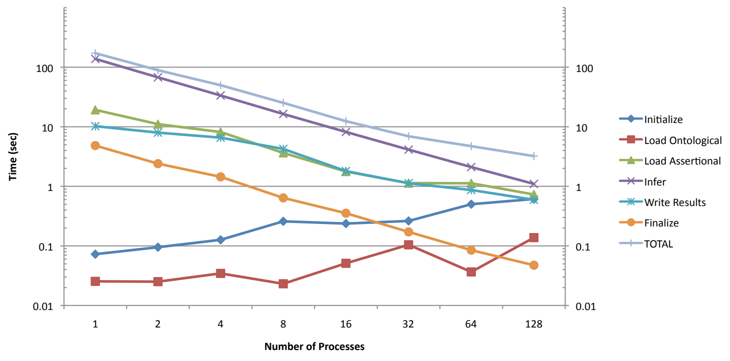

Fig. 3. Breakdown of times for computing finite RDFS closure on LUBM10k/1024 for varying number of processes.

5.2 Inferences Produced

As mentioned in Section 5, our implementation does not eliminate duplicate triples in different partitions; instead, for performance, we simply have each process write all of its triples to separate files. Therefore, we present an evaluation on how many duplicate inferences are produced by this approach as we scale across number of processes and data set size. When run with one process, no duplicates are created, and so we use the number of inferences from one process as our standard for comparison. However, only the three smallest data sets could be run on one process, so we provide evaluation only for those three data sets. For evaluation purposes, we count the axiomatic triples as inferences since they are added during the inferencing process, and the axiomatic triples are duplicated on all partitions.

Data sets	# Assertions	Inferences for Varying Number of Processes							
		1	2	4	8	16	32	64	128
10k/1024	1,349,680	1,200,186	1,205,309	1,215,685	1,235,831	1,274,320	1,339,677	1,446,088	1,621,892
10k/512	2,699,360	2,397,066	2,402,267	2,412,473	2,433,087	2,473,273	2,549,186	2,678,194	2,888,254
10k/256	5,398,720	4,785,083	4,790,285	4,800,510	4,821,327	4,861,330	4,940,744	5,091,493	5,348,844
10k/128	10,797,441		9,572,280	9,582,529	9,603,158	9,644,372	9,725,863	9,885,081	10,184,984
10k/64	21,594,882				19,169,424	19,210,312	19,291,708	19,453,885	19,774,980
10k/32	43,189,765					38,337,401	38,420,096	38,583,284	38,906,839
10k/16	86,379,530						76,671,931	76,835,141	77,163,076
10k/8	172,759,061							153,348,394	153,676,871
10k/4	345,518,123								306,700,784

Table 1. Information about data sets and inferences produced. Note that inferences are unique only for a single process.

Table 1 shows the number of inferences produced for different data sets and number of processes. Note that only the numbers of inferences for one process are

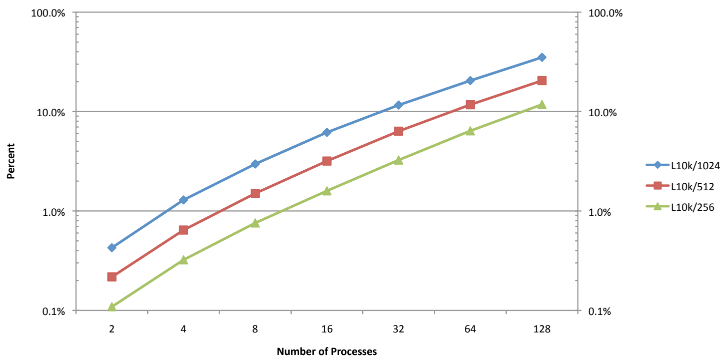

Fig. 4. The percentage of inferences duplicated for the three smallest data sets for varying number of processes.

unique inferences only (no duplicates). In Figure 4, the percentage of duplicate inferences grows super-linearly at first (across processes) but soon begins to taper off. As the data set doubles in size, the percentage of duplicates seems to halve; this indicates that duplicates result from fairly static information that somehow ends up in all partitions. This would include ontological data, axiomatic triples, and inferences that proliferate fairly easily. The rules that created the most duplicate inferences are those that most users would probably choose to exclude such as *rdfs4a/b* (everything is a rdfs:Resource) and *rdfs1* (all literals are rdfs:Literals). The exception, though, is *rdfs9* which infers rdf:type triples from subclass hierarchy. Interestingly, the only rule for which there were any inferences and no duplicate inferences was *rdfs7*, inferring statements from subproperty hierarchy.

6 Future Work

We see two general directions for future work: scalability and expressivity.

Scalability can be improved by using more efficient in-memory representations of RDF data and by scaling to more processes. In the former case, we hope to employ BitMat [19], a very compact in-memory representation of RDF that also allows for efficient, basic graph pattern querying. In the latter case, we would like to move our code to a more scalable environment like the Blue Gene/L (BG/L) at RPI's CCNI. Our system could also be extended to use a pipelining approach allowing us to scale to data sets which are limited only by disk space. We could simply read in as much of the assertional triples as will fit in a single process, perform inferencing, write results, and then request more assertional triples from disk.

Expressivity can be improved by simply adding more rules that fit the M1 classes described thus far and also by discovering more classes of rules that are

APS. In the former case, we have already identified a handful of OWL2RL[14] rules that can be supported in this paradigm (e.g., symmetric properties, equivalent classes, has-value restrictions, etc.), although a multiple-pass approach may be needed instead of the single pass approach in Algorithm 1. We plan to add support for these features in the near future. In the latter case, we are investigating with other colleagues how to handle joins among triple patterns in such a cluster environment so that we can handle non-M1 rules.

We also believe that this approach my be useful in a MapReduce implementation of an RDFS reasoner. In the map phase, ontological triples could be mapped to all reducers and assertional triples to only one reducer each; then, in the reduce phase, the rules can be applied to all the triples mapped to that partition. Of course, considerations may have to be made to ensure that the partitions are not overloaded with too many triples.

7 Conclusion

We have defined a partitioning scheme—abox partitioning—and five classes of rules which can be used to perform complete parallel inferencing on abox partitions. We showed that all of the finite RDFS rules are abox partitioning safe and derived an embarrassingly parallel algorithm for producing the finite RDFS closure. We implemented a C/MPI version of the algorithm and performed an evaluation on a cluster of large memory Opteron machines that showed linear scaling for the inferencing time. We also showed that although some inferences are duplicated, the percentage is generally small for larger data sets. Our results exceed the results reported by related work in that we scale to 128 processes, producing a closure of roughly 650 million triples from an initial data set of roughly 345 million triples in only 8 minutes and 25 seconds, for which the actual inferencing time was only 4 minutes and 51 seconds. To our knowledge, no other system exists that can produce the (finite) RDFS closure on such large data sets in so little time.

Acknowledgements. We thank Medha Atre and Gregory Todd Williams for their insightful comments in reviewing this paper.

References

1. Hayes, P.: RDF Semantics. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ (2004)
2. Wilkinson, B., Allen, M.: 3. In: Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. 2 edn. Prentice Hall (2005)
3. Anadiotis, G., Kotoulas, S., Oren, E., Siebes, R., van Harmelen, F., Drost, N., Kemp, R., Maassen, J., Seinstra, F.J., Bal, H.E.: MaRVIN: a distributed platform for massive RDF inference. http://www.larkc.eu/marvin/btc2008.pdf (2008)

[14] http://www.w3.org/TR/2009/WD-owl2-profiles-20090421/#OWL_2_RL

4. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: MaRVIN: A platform for large-scale analysis of Semantic Web data. In: Proceeding of the WebSci'09: Society On-Line. (March 2009)
5. Soma, R., Prasanna, V.K.: A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases. Technical report, University of Southern California (2008)
6. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington DC, USA, IEEE Computer Society (2008) 75–82
7. ter Horst, H.J.: Combining RDF and part of OWL with rules: Semantics, decidability, complexity. In: International Semantic Web Conference. (2005) 668–684
8. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2-3) (October 2005) 158–182
9. Fang, Q., Zhao, Y., Yang, G., Zheng, W.: Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning. In: The Semantic Web. Volume 5367/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 91–105
10. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS Reasoning and Query Answering on Top of DHTs. In: The Semantic Web - ISWC 2008. Volume 5318/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 499–516
11. Oren, E., Gueret, C., Schlobach, S.: Anytime Query Answering in RDF through Evolutionary Algorithms. In: The Semantic Web - ISWC 2008. Volume 5318/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 98–113
12. Rudolph, S., Tserendorj, T., Hitzler, P.: What is Approximate Reasoning? In: Web Reasoning and Rule Systems. Volume 5341/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 150–164
13. Bock, J.: Parallel Computation Techniques for Ontology Reasoning. In: The Semantic Web - ISWC 2008. Volume 5318/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 901–906
14. Liebig, T., Muller, F.: Parallelizing Tableaux-Based Description Logic Reasoning. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops. Volume 4806/2007 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2007) 1135–1144
15. Hogan, A., Harth, A., Polleres, A.: Scalable Authoritative OWL Reasoning on a Billion Triples. In: Proceedings of Billion Triple Semantic Web Challenge. (2008)
16. Hogan, A., Harth, A., Polleres, A.: SAOR: Authoritative Reasoning for the Web. In: ASWC. (2008) 76–90
17. Hogan, A., Harth, A., Polleres, A.: Scalable Authoritative OWL Reasoning for the Web. International Journal on Semantic Web and Information Systems (2009)
18. Beckett, D.: RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/ (2004)
19. Atre, M., Srinivasan, J., Hendler, J.A.: BitMat: A Main Memory RDF Triple Store. Technical report, Rensselaer Polytechnic Institute (January 2009)