Scalable In-memory RDFS Closure on Billions of Triples

Eric L. Goodman[1] and David Mizell[2]

[1] Sandia National Laboratories, Albuquerque, NM, USA
elgoodm@sandia.gov
[2] Cray, Inc., Seattle, WA, USA
dmizell@cray.com

Abstract. We present an RDFS closure algorithm, specifically designed and implemented on the Cray XMT supercomputer, that obtains inference rates of 13 million inferences per second on the largest system configuration we used. The Cray XMT, with its large global memory (4TB for our experiments), permits the construction of a conceptually straightforward algorithm, fundamentally a series of operations on a shared hash table. Each thread is given a partition of triple data to process, a dedicated copy of the ontology to apply to the data, and a reference to the hash table into which it inserts inferred triples. The global nature of the hash table allows the algorithm to avoid a common obstacle for distributed memory machines: the creation of duplicate triples. On LUBM data sets ranging between 1.3 billion and 5.3 billion triples, we obtain nearly linear speedup except for two portions: file I/O, which can be ameliorated with the additional service nodes, and data structure initialization, which requires nearly constant time for runs involving 32 processors or more.

Keywords: Semantic Web, RDFS Closure, Cray XMT, Hashing

1 Introduction

Semantic web data in its most common format, the Resource Description Framework (RDF), has two primary means for defining ontologies: RDF Schema (RDFS) and the Web Ontology Language (OWL). Ontologies are useful mechanisms for organizing domain knowledge, allowing explicit, formal descriptions of data to be defined and shared. Also, ontologies allow reasoning about the domain, exposing new facts through application of the ontology to existing data.

In this paper we examine the simpler of the ontology languages, RDFS, and perform RDFS reasoning on billions of triples completely in-memory on a highly multithreaded shared-memory supercomputer, the Cray XMT. To our knowledge, no one has performed such large RDFS reasoning in a single global shared address space, nor has anyone previously achieved the inferencing rates we report in this paper.

The rest of the paper is organized as follows. Section 2 describes the Cray XMT, its unique characteristics, and why it may be well suited to RDFS closure

and many semantic web applications in general. Section 3 describes the algorithm we employ to perform closure. Sections 4 and 5 describe the experimental setup and the results. We then conclude in sections 6 and 7 with a comparison to other approaches and our path forward.

2 Cray XMT

The Cray XMT is a unique shared-memory machine with multithreaded processors especially designed to support fine-grained parallelism and perform well despite memory and network latency. Each of the custom-designed compute processors (called *Threadstorm* processors) comes equipped with 128 hardware threads, called *streams* in XMT parlance, and the processor instead of the operating system has responsibility for scheduling the streams. To allow for single-cycle context switching, each stream has a program counter, a status word, eight target registers, and thirty-two general purpose registers. At each instruction cycle, an instruction issued by one stream is moved into the execution pipeline. The large number of streams allows each processor to avoid stalls due to memory requests to a much larger extent than commodity microprocessors. For example, after a processor has processed an instruction for one stream, it can cycle through the other streams before returning to the original one, by which time some requests to memory may have completed. Each *Threadstorm* processor can currently support 8 GB of memory per processor, all of which is globally accessible. The system we use in this study has 512 processors and 4 TB of shared memory. We also employed 16 Opteron nodes of the service partition, which directly perform file and network I/O on behalf of the compute processors.

Programming on the XMT consists of writing C/C++ code augmented with non-standard language features including generics, intrinsics, futures, and performance-tuning compiler directives such as pragmas.

Generics are a set of functions the Cray XMT compiler supports that operate atomically on scalar values, performing either `read`, `write`, `purge`, `touch`, and `int_fetch_add` operations. Each 8-byte word of memory is associated with a full/empty bit and the read and write operations interact with these bits to provide light-weight synchronization between threads. Here are some examples of the generics provided:

- *readxx*: Returns the value of a variable without checking the full-empty bit.
- *readfe*: Returns the value of a variable when the variable is in a full state, and simultaneously sets the bit to be empty.
- *writeef*: Writes a value to a variable if the variable is in the empty state, and simultaneously sets the bit to be full.
- *int_fetch_add*: Atomically adds an integer value to a variable.

Besides generics, there are also intrinsic functions that expose many low-level machine operations to the C/C++ programmer.

Parallelism is achieved explicitly through the use of futures, or implicitly, when the complier attempts to automatically parallelize for loops. Futures allow programmers to explicitly launch threads to perform some function. Besides

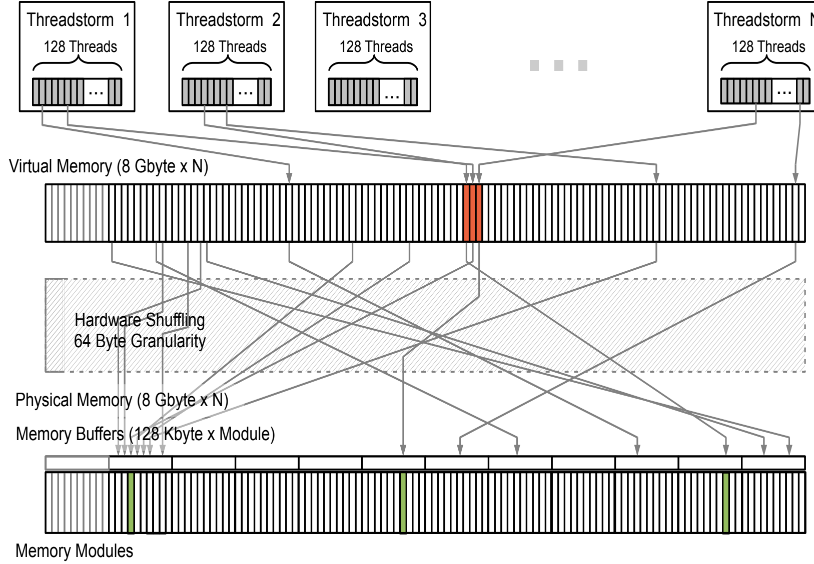


Fig. 1. Cray XMT Threadstorm memory subsystem: Threadstorm processors access a virtual memory address space that is mapped through hardware shuffling to actual physical locations to mitigate memory access hotspots.

explicit parallelism through futures, the compiler attempts to automatically parallelize for loops, enabling implicit parallelism. The programmer can also provide pragmas that provide hints to the compiler on how to schedule iterations of the for loop to various threads, whether it be by blocks, interleaved, or dynamically, or supply hints on how many streams to use per processor, etc. We extensively use the `#pragma mta for all streams i of n` construct that allows programmers to be cognizant of the total number of streams that the runtime has assigned to the loop, as well as providing an iteration index that can be treated as the id of the stream assigned to each iteration.

2.1 Applicability of the XMT to the Semantic Web

The XMT and its predecessors have a significant history of performing quite well on graph algorithms and on applications that can be posed as graph problems. Bader and Madduri [2] report impressive execution times and speedup for fundamental graph theory problems, breadth-first search and *st*-connectivity. Later, Madduri et al. [8] use the Δ-stepping algorithm, performing single source shortest path on a scale-free, billion-edge graph in less than ten seconds on 40 processors. More recently, Chin et al. [3] examine triad census algorithms for social network analysis on graphs with edges in the hundreds of millions. Also, Jin et al. [6] perform power grid contingency analysis on the XMT by posing the

problem as finding important links using betweeness centrality of the edges as a metric.

To see how this relates to the semantic web, consider that the semantic web amounts to a large, sparse graph. The predicate of an RDF triple can be thought of as a directed edge between two nodes, the subject and the object. Several have proposed thinking of the semantic web as a graph, and have suggested extensions to SPARQL to enable advanced graph queries otherwise not available in standard SPARQL, including SPARQ2L [1], nSPARQL [10], and SPARQLeR [7]. Also, Stocker et al. [12] present an optimization scheme for SPARQL queries based on the graph nature of RDF data. They state one limitation of the approach is due to its implementation in main memory. However, this is not as much of an issue for the XMT with its scalable global shared memory. Indeed, for the machine we use in this study, its 4 TB of shared memory is sufficient for the data of many semantic web applications to reside completely in memory. The XMT is well suited for applications with (a) abundant parallelism, and (b) little or no locality of reference. Many graph problems fit this description, and we expect that some semantic web applications will as well, especially those that involve more complex queries and inferencing.

3 Algorithm

The algorithm we present here performs incomplete RDFS reasoning, and calculates only the subset of rules that require two antecedents, namely rdfs rules 2, 3, 5, 7, 9, and 11 (for reference, see Table 1). This is fairly common in practice, as the other rules are easily implemented and parallelized and generally produce results that are not often used or can be inferred later at run time for individual queries. The flow of the algorithm (see Figure 2) is similar to the one outlined by Urbani et al. [13] in that only a single pass over the rule set is required (excluding rare cases where the ontology operates on RDFS properties) for full and correct inferencing. However, we move processing of the transitivity rules, 5 and 11, to the beginning. These rules accept no inputs except for existing ontological triples, and thus can be executed safely at the beginning. Also, we do not remove duplicates because our algorithm produces no duplicates. We use a global hash table described by Goodman et al. [4] to store the original and inferred triples. Thus, duplicate removal is accomplished *in-situ*, i.e. insertion of any triples produced during inferencing that already exists in the table results in a no-op.

As a preprocessing step, we run the set of triples through a dictionary encoding that translates the resource and predicate strings into 64 bit integers. We again use the hash table described earlier to store the mapping and perform the translation. As the triples are now sets of three integers, we can make use of the following hash function during the RDFS closure algorithm:

$$\text{hash}(t) = ((t.s + t.p \cdot B + t.o \cdot B^2) \cdot C) \mod S_{table} \quad (1)$$

where t is a triple, $C = 31,280,644,937,747$ is a large prime constant (taken from [4]), S_{table} is the size of the hash table, and B is a base to prevent the sub-

Rule	Condition 1	Condition 2 (optional)	Triple to Add
lg	s p o (o is a literal)		s p _:n
gl	s p _:n		s p o
rdf1	s p o		p type Property
rdf2	s p o (o is a literal of type t)		_:n type t
rdfs1	s p o (o is a literal)		_:n type Literal
rdfs2	**p domain x**	**s p o**	**s type x**
rdfs3	**p range x**	**s p o**	**o type x**
rdfs4a	s p o		s type Resource
rdfs4b	s p o		o type Resource
rdfs5	**p subPropertyOf q**	**q subPropertyOf r**	**p subPropertyOf r**
rdfs6	p type Property		p subPropertyOf p
rdfs7	**s p o**	**p subPropertyOf q**	**s q o**
rdfs8	s type Class		s subClassOf Resource
rdfs9	**s type x**	**x subClassOf y**	**s type y**
rdfs10	s type Class		s subClassOf s
rdfs11	**x subClassOf y**	**y subClassOf z**	**x subClassOf z**
rdfs12	p type Container-MembershipProperty		p subPropertyOf member
rdfs13	o type Datatype		o subClassOf Literal

Table 1. This table lists all the rules that are part of RDFS entailment [11]. The rules in bold are the ones we implemented for our closure algorithm.

Algorithm: RDFS Closure

1: Read data from service nodes
2: Create and populate ontology data structures
3:    Create and populate multimaps
4:    Apply transitivity rules, **rdfs5** and **rdfs11**
5:    Replicate multimap data structures.
6: Insert original triples into hash table
7: Add matching triples to queues
8: **rdfs7** - Subproperty Inheritance
9:    Add matching triples to domain and range queues
10: **rdfs2** - Domain
11:    Add matching triples to subclass queue
12: **rdfs3** - Range
13:    Add matching triples to subclass queue
14: **rdfs9** - Subclass Inheritance

Fig. 2. Overview of RDFS Closure algorithm on the XMT.

ject, predicate, and object ids from colliding. The current dictionary encoding algorithm assigns ids to resources and properties by keeping a running counter, and a triple element's id is the value of the counter at the time the element was seen during processing. Thus, the set of integers assigned to the subject, predicate, and objects overlap. The base helps the hash function to avoid collisions due to this overlap. For our experiments, we used $B = 5e9$, near the maximum value assigned to any triple element. This hash function was decided upon after some experimentation, but probably deserves further attention.

The first step (line 1 of Figure 2) of the RDFS closure process on the XMT is to transfer the RDF triple data from the service nodes to the compute nodes. We use the Cray XMT snapshot library that allows programs to move data back and forth from the Lustre file system. On the service nodes a Linux process called a file service worker (fsworker) coordinates the movement of data from disk to the compute nodes. Multiple file service workers can run on multiple service nodes, providing greater aggregate bandwidth.

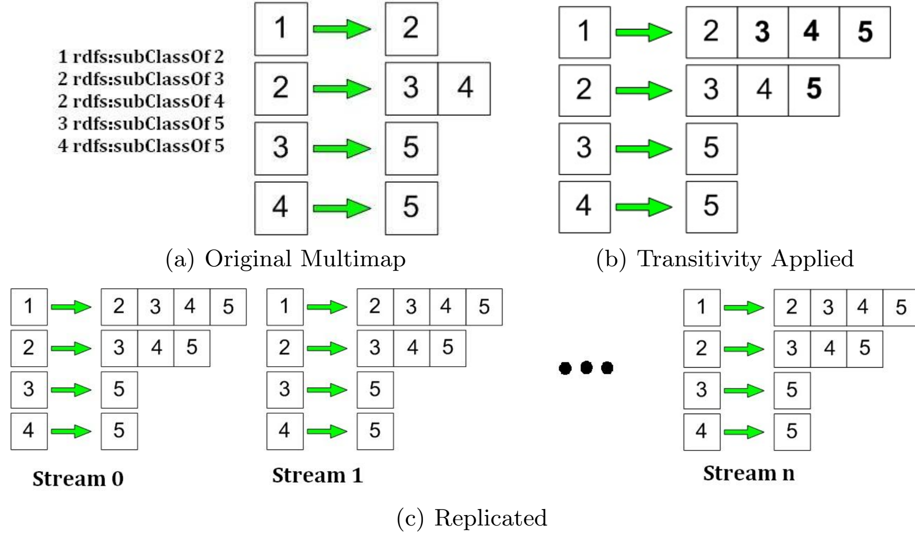
The next step (lines 2-5 of Figure 2) is to load the ontological data into multimap data structures. All of the rules under consideration involve a join between a

- data triple or an ontological triple with an
- ontological triple,

and in all cases, the subject of the latter ontological triple is matched with a subject, predicate, or object of the former. To speed processing of the rules and the associated join operations, we create four multimaps of the form $f : Z \rightarrow Z^*$, one for each of the predicates **rdfs:subPropertyOf**, **rdfs:domain**, **rdfs:range**, and **rdfs:subClassOf**, that maps subject ids to potentially multiple object ids. For example, the **rdfs:subClassOf** multimap is indexed according to class resources and maps to all stated super classes of those resources. After initially populating the multimap with known mappings as asserted in the original triple set (see Figure 3(a)), we then update each of the **rdfs:subPropertyOf** and **rdfs:subClassOf** multimaps with a complete listing of all super properties and super classes as dictated by inference rules 5 and 11 (see Figure 3(b)).

After populating the multimaps and applying the transitivity rules, we then replicate the entirety of this data and the accompanying data structures for each and every stream (see Figure 3(c)). The reason for this is to avoid read-hotspotting. The ontology is relatively small compared with the data to be processed, so many streams end up vying for access if only one copy of the ontology is available to all streams. Replicating the data circumvents these memory contention issues.

The next two steps (lines 6-7 of Figure 2) involve iterating over all of the original triples and adding them to the hash table, and also populating four queues, also implemented as hash tables, that store triples matching rules 2, 3, 7, and 9. Ideally, this could be solved with one pass through the original triple set, but the XMT favors tight, succinct loops. Processing all steps at once results in poor scalability, probably because the compiler is better able to optimize smaller, simpler loops. Thus, each of these individual steps receives its own for loop.



(a) Original Multimap (b) Transitivity Applied

(c) Replicated

Fig. 3. Subfigure (a) shows a set of triples containing *rdfs:subclassOf* information. The integers represent URIs. The displayed mulitmap represents the state after Step *3* of the RDFS Closure algorithm. Subfigure (b) shows the state after Step *4*, where each class has a complete listing of all superclasses. Subfigure (c) illustrates that after Step *5*, all of the multimaps are replicated for dedicated stream usage.

The last four steps (lines 8-14 of Figure 2) complete the calculation of RDFS closure. In general, each of these steps follow the `ComputeRule` procedure outlined in Figure 4. There are three parameters: an array of triples that match the given rule, a multimap that stores subject/object mappings for a given predicate, and a buffer to store new triples. We use the **`for all streams`** construct, allowing us to know the total number of streams, *num_streams*, and a stream identifier, *stream_id* to use within the outer for loop. Line 5 partitions the matching triples into nearly equal-sized chunks for each stream. The for loop from lines 7 to 10 determines the number new triples that will be created. This first pass at the data allows us to call `int_fetch_add` once for each stream instead of l times, where l is the length of the matching triple array. In essence, when we call `int_fetch_add` in line 11 we claim a block of indices in the buffer array that can be iterated over locally instead of requiring global cooperation by all streams. This helps to avoid hot-spots on the *total* variable. Lines 12 through 17 iterate over the stream's set of triples and then iterate over the set of matching rules in the multimap to create new inferred triples. After application of `ComputeRule`, inferred triples in the buffer are added both to the main hash table and to appropriate rule queues. Again, these two steps are done separately to maintain scalability.

Overall, the scalability and success of this algorithm largely rests on three main factors:

```
 1: procedure ComputeRule(matching_triples, multimap, buffer)
 2:     total ← 0
 3:     for all streams stream_id of num_streams do
 4:         l ← length(matching_triples)
 5:         beg, end ← determine_beg_end(l, num_streams, stream_id)
 6:         local_total ← 0
 7:         for i ← beg, end do
 8:             num ← num_values(matching_triples[i], multimap)
 9:             local_total ← local_total + num
10:         end for
11:         start = int_fetch_add(total, local_total)
12:         for i ← beg, end do
13:             for j ← 1, num_values(matching_triples[i]) do
14:                 buffer[start] ← new_triple(matching_triples[i], multimap, j)
15:                 start ← start + 1
16:             end for
17:         end for
18:     end for
19: end procedure
```

Fig. 4. This figure outlines the general process for computing RDFS rules 2, 3, 7, and 9.

- Extensive use of hash tables: Hash tables proved effective for the implementation of RDFS closure by allowing the algorithm to be conceptually simple and to maintain scalability over large numbers of processors. The global nature of the hash table also removed the time-consuming deduplication phase, a necessary step for many algorithms targeting distributed memory platforms.
- Multiple tight loops: We favored small succinct loops, even if that meant iterating over the same data twice, as it allowed scalable performance.
- Replicating the ontology: A single instance of the ontology created a hotspot as many streams were accessing the same data simultaneously. Replicating the ontology removed memory contention issues. This is likely generalizable to other situations when many streams must use small, read-only data structures.

4 Data Sets

We used the Lehigh University Benchmark (LUBM) [5]. Specifically, we generated the LUBM10k and LUBM40k data sets, approximately 1.33 and 5.34 billion triples, respectively. Both increase in size after closure by about 25%, resulting in 341 million and 1.36 billion inferences.

To validate our approach, we applied a fixed-point algorithm written in python[3] to smaller LUBM instances, commenting out rules we did not implement in our algorithm.

The code we wrote for RDFS closure on the XMT is open source and publically available. Most of the code is published as part of the MapReduceXMT code base[4]. Our initial implementation first looked at porting the MapReduce implementation of Urbani, et al. [13] to the XMT, but to obtain better performance we adopted an implementation that was more closely tailored for the XMT. The main class is `rdfs_closure_native.cpp` located in the `test` directory. We also made use of functionality provided by the MultiThreaded Graph Library[5], primarily `mtgl/xmt_hash_table.hpp`.

5 Experiments

Figure 5 shows the times for computing RDFS closure on LUBM10k and LUBM40k. For all runs we use 16 service nodes each running one fsworker. We conducted runs using between 32 and 512 processors. We did not go smaller than 32 because the snapshot libraries require at least $2f$ processors, where f is the number of fsworkers.

The file I/O times reported below include only the time to read in the file. As the XMT's most common use case will likely be a memory-resident semantic database, we include only the load time as any writes will likely be asynchronous.

Also, the initialization times reported below only include the time for the construction of the main hash table, the four smaller queues, and two large triple buffers. Other data structure initialization calls were made, but they were significantly smaller and included in times of the other phases.

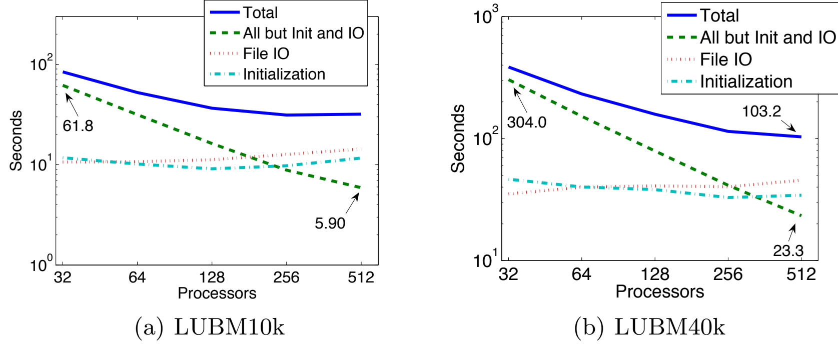
The algorithm shows good scalability for everything but file I/O and initialization, which are nearly constant for a given problem size and irrespective of the number of processors. Comparing the times of the 32 and 512 processor runs on LUBM40k, we achieve about 13 times speedup for the non-I/O/init portion, where linear speedup in this processor range is 16. On LUBM10k, the speedup is lower, about 10, indicating that the problem size of LUBM10k is not sufficient to utilize the full 512 processor system as well as LUBM40k.

Figure 6 shows the inference rate in inferences/second each method obtained for each of the data sets. Initially, better rates are achieved on LUBM10k for smaller numbers of processors. However, for large processor counts, the algorithm's better scalability on LUBM40k wins out, obtaining a rate of 13.2 million inferences per second with 512 processors.

[3] http://www.ivan-herman.net/Misc/PythonStuff/RDFSClosure/Doc/RDFSClosure-module.html

[4] https://software.sandia.gov/trac/MapReduceXMT

[5] https://software.sandia.gov/trac/mtgl



(a) LUBM10k (b) LUBM40k

Fig. 5. Times recorded for LUBM10k and LUBM40k.

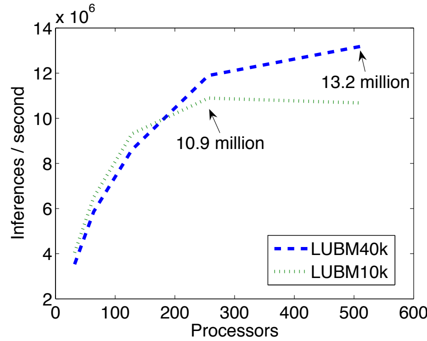


Fig. 6. Inference rates obtained on LUBM10k and LUBM40k.

5.1 Modifying the number of service nodes

We also examined how the number of fsworkers affects aggregate I/O throughput to the compute nodes. We kept the number of processors constant at 128 and varied the number of fsworkers between 1 and 16, each fsworker running on a single node of the service partition. We tested the read time on LUBM40k. Figure 7 shows the rates obtained and the ideal rates expected if performance increased linearly with the number of fsworkers. Overall, we experienced about a 12.4 increase in I/O performance for 16 fsworkers over 1 fsworkers, indicating that if I/O is a bottleneck for particular applications, it can be ameliorated with additional service nodes and fsworkers.

6 Comparison to other Approaches

There are two main candidates for comparison with our approach. The first is an MPI-based implementation of RDFS closure developed by Weaver and Hendler [16], which we will refer to simply as *MPI*. The second is WebPIE, a MapReduce style computation developed by Urbani et al. The authors first focused on

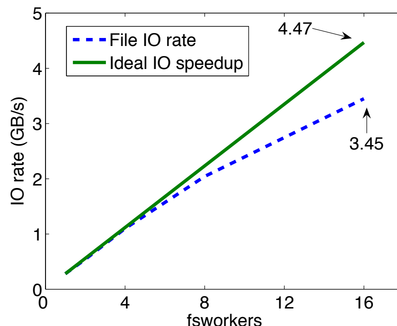


Fig. 7. I/O rates achieved for various number of fsworkers.

RDFS closure [13] and then expanded their efforts to include a fragment of OWL semantics [14].

Weaver and Hendler developed an embarrassingly parallel algorithm for RDFS closure by replicating the ontology for each process and partitioning up the data. They utilized a 32 node machine with two dual-core 2.6 GHz Opteron processors per node. They achieved linear scalability but with the side effect of producing duplicates. The individual processes do not communicate and do not share data. Thus, it is impossible to know if duplicates are being produced.

Besides the fundamental algorithmic difference between our implementation and theirs concerning duplicate creation (they create duplicates when we do not), there are several other factors which make a direct comparison difficult. For one, they operate on raw triple data rather than integers. Also, they perform all of the rules listed in Table 1 while we compute a subset. Finally, they include both reading the data in from disk and writing the result out, where we only report the read time.

The work of Urbani et al. offers a much more direct comparison to our work, as they first perform a dictionary encoding algorithm [15] as a preprocessing step and also perform a similar subset of RDFS rules as we do in this work. They employ a series of a MapReduce functions to perform RDFS closure. Like the Weaver and Hendler approach, their algorithm produces duplicates during processing; however, they account for that and have a specific stage dedicated for removal of duplicates. Their experiments were conducted on a 64 node machine with two dual-core 2.4GHz Opteron processors per node. It should be noted that they've shown results computing closure on 100 billion triples. Our approach, since we limit ourselves to in-memory computations, cannot reach that scale, at least given a similar number of processors. With a specialized version of the hash table code with memory optimizations not available in the more general implementation, we estimate the largest data set we could handle within 4 TB of memory to be about 20 billion triples. However, the next generation XMT system, due around the end of 2010, is expected to have as much as eight times as much memory per processor.

	Inferences/second
MPI (32 nodes)	$574e3$
WebPIE (64 nodes)	$\sim 700e3$
XMT (512 proc.)	$13.2e6$
XMT (256 proc.)	$11.9e6$
XMT (128 proc.)	$9.43e6$
XMT (64 proc.)	$6.50e6$
XMT (32 proc.)	$4.04e6$

Table 2. Listed above are the best rates achieved on LUBM data sets for various platforms, sizes, and approaches. MPI is the Weaver and Hendler approach [16]. The WebPIE number is an estimate for the RDFS closure portion of a larger OWL Horst calculation [14].

	Threadstorm processors to nodes
MPI	7.05
WebPIE	9.28

Table 3. This table reports the speedup achieved by the RDFS closure algorithm on the XMT versus the two top competing approaches.

Table 2 shows the rate of inferences/second each method achieves for varying platforms sizes. Table 3 shows the speedup in inferences/second we obtained over these other methods, again with the caveat that this comparison is inexact due to differences described above. We compare the number of *Threadstorm* processors to an equal number of Opteron cores (i.e. a socket-to-socket comparison). For both cases we compare against results on LUBM, though the generated sizes vary from 345 million to 100 billion. The numbers on LUBM for Urbani we take from their OWL paper [14], using the fact that the first computation in OWL inferencing for their algorithm is an RDFS closure. Since RDFS closure on LUBM data sets, regardless of size, generally produce $\frac{X}{4}$ inferences, where X is the original number of triples (at least for the set of rules in bold in Table 1), and using times reported in conversations with the authors for the initial RDFS closure stage, we calculate the estimated inference rate for this stage of their algorithm to be about 700 thousand inferences a second.

7 Conclusions and Future Work

We have presented an RDFS closure algorithm for the Cray XMT that achieves inference rates far beyond that of the current state of the art, about 7-9 times faster when comparing *Threadstorm* processors to commodity processor cores. Part of our approach's success is due to the unique architecture of the Cray XMT, allowing us to store large triple stores entirely in memory, avoiding duplication of triples and also costly trips to disk.

There are several obvious avenues for future work. In this paper we examined only artificially generated data sets with a relatively simple ontology. We would like to examine real-world data, and especially explore the effect that more complex ontologies have on our algorithm and make adjustments as necessary.

We also want to expand this work to encompass OWL semantics, probably focusing first on the Horst fragment [9], as it is among the most computationally feasible subsets of OWL. We believe the XMT will again be well suited to this problem, maybe even more so than RDFS. For instance, some rules in OWL Horst require two instance triples to be joined with one schema triple. This poses a challenge for distributed memory machines in that you can no longer easily partition the data as with RDFS, which had at most one instance triple as an antecedent. The XMT can store all of the instance triples in its large global memory, and create indices as needed to quickly find matching triples.

Also, there are some tricks to try that might help eliminate some of the constant overhead we see with file I/O (for a fixed number of service nodes) and data structure initialization. File I/O and initialization both require low numbers of threads, so it may be possible to overlay them, using a future to explicitly launch initialization just before loading the data.

One limitation of the approach presented in this paper is the use of fixed table sizes, forcing the user to have accurate estimates of the number inferences that may be generated. Using Hashing with Chaining and Region-based Memory Allocation (HACHAR), also presented in [4], may alleviate this requirement. It allows for low-cost dynamic growth of the hash table and permits load factors far in excess of the initial size; however, the current implementation exhibits poorer scalability than the open addressing scheme we used for this paper.

Finally, we wish to explore a fairer comparison between our work on a shared memory platform to a similar in-memory approach on distributed memory machines. The work presented by Weaver and Hendler [16] is the closest we can find to in-memory RDFS closure on a cluster, but the algorithmic differences make it difficult to draw definite conclusions. The *MapReduce-MPI Library* [6], a MapReduce inspired framework built on top of MPI, permits the development of entirely in-memory algorithms using standard MapReduce constructs. As such, it is an ideal candidate for producing a more comparable RDFS closure implementation for distributed memory platforms.

Acknowledgments. This research was funded by the DoD via the CASS-MT program at Pacific Northwest National Laboratory. We wish to thank the Cray XMT software development team and their manager, Mike McCardle, for providing us access to *Nemo*, the 512-processor XMT in Cray's development lab. We also thank Greg Mackey and Sinan al-Saffar for their thoughtful reviews of this paper.

[6] `http://www.sandia.gov/~sjplimp/mapreduce.html`

References

1. Anyanwu, K., Maduko, A., Sheth, A.P. SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In Proceedings of WWW. Banff, Alberta, Canada (2007)
2. Bader, D.A., Madduri, K. Designing Multithreaded Algorithms for Breadth-First Search and *st*-connectivity on the Cray MTA-2. In Proceedings of the 35th International Conference on Parallel Processing. Columbus, OH, USA (2006)
3. Chin, G., Marquez, A., Choudhury, S., Maschoff, K. Implementing and Evaluating Multithreaded Triad Census Algorithms on the Cray XMT. In Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing. Rome, Italy (2009)
4. Goodman, E.L., Haglin D.J., Scherrer, C., Chavarría-Miranda, D., Mogill, J., Feo J. Hashing Strategies for the Cray XMT. In Proceedings of the IEEE Workshop on Multi-Threaded Architectures and Applications. Atlanta, GA, USA (2010)
5. 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
6. Jin, S., Huang, Z., Chen, Y., Chavarría, D.G., Feo, J., Wong, P.C. A Novel Application of Parallel Betweeness Centrality to Power Grid Contingency Analysis. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium. Atlanta, GA (2010)
7. Kochut, K.J., Janik, M. SPARQLeR: Extended Sparql for Semantic Association Discovery. In Proceedings of the 4th European Semantic Web Conference. Innsbruck, Austria (2007)
8. Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In Proceedings of the Workshop on Algorithm Engineering and Experiments. New Orleans, LA, USA (2007)
9. ter Horst, H.J. Completeness, Decidability, and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL Vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web 3(2-3) (October 2005) 79-115.
10. Pérez, J., Arenas, M., Guiterrez, C. nSPARQL: A Navigational Language for RDF. In Proceedings of the 7th International Semantic Web Conference. Springer, Karlsruhe, Germany (2008)
11. RDF Semantics, W3C Recommendation, 10 Februrary 2004, `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In Proceedings of WWW. Beijing, China (2008)
13. Urbani, J., Kotoulas S., Oren, E., van Harmelen, F. Scalable Distributed Reasoning using MapReduce. In Proceedings of the 8th International Semantic Web Conference. Springer, Washington D.C., USA (2009)
14. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In Proceedings of the 7th Extended Semantic Web Conference. Heraklion, Greece (2010)
15. Urbani, J., Maaseen, J., Bal, H. Massive Semantic Web data compression with MapReduce. In Proceedings of the MapReduce workshop at High Performance Distributed Computing Symposium. Chicago, IL, USA (2010)

16. Weaver, J., Hendler, J.A. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In Proceedings of 8th International Semantic Web Conference. Washington, DC, USA (2009)