2015 IEEE International Conference on Cluster Computing

High-performance, Distributed Dictionary Encoding of RDF Datasets

Alessandro Morari*, Jesse Weaver*, Oreste Villa†, David Haglin*,
Antonino Tumeo*, Vito Giovanni Castellana*, John Feo*
*Pacific Northwest National Laboratory, Richland, Washington 99354
Email: {alessandro.morari,jesse.weaver,david.haglin,antonino.tumeo,vitoGiovanni.castellana, john.feo}@pnnl.gov
†NVIDIA Research, Santa Clara, California 95051
Email: ovilla@nvidia.com

***Abstract*—In this work we propose a novel approach for RDF (Resource Description Framework) dictionary encoding that employs a parallel RDF parser and a distributed dictionary data structure, exploiting RDF-specific optimizations. In contrast with previous solutions, this approach exploits the Partitioned Global Address Space (PGAS) programming model combined with active messages. We evaluate the performance of our dictionary encoder in our RDF database, GEMS (Graph Engine for Multithreaded Systems), and provide an empirical comparison against previous approaches. Our comparison shows that our dictionary encoder scales significantly better and achieves higher performance than the current state of the art, providing a key element for the realization of a more efficient RDF database.**

I. Introduction

The Resource Description Framework (RDF) [1] is a flexible data model recommended by the World Wide Web Consortium (W3C) in realizing the Semantic Web. RDF organizes data in triples, sequences of subject-object-predicate terms that naturally maps to a labeled graph. There has been significant uptake of RDF for publishing/integrating data as Linked Data [2] in various communities including: open government; biomedical research; environmental sciences; humanities; social networks; geography; tourism; and healthcare. Perhaps the most significant uptake of RDF is in the recommendation and use of schema.org annotations by major search engines including Bing, Google, and Yahoo!. RDF data are periodically crawled and aggregated into datasets over which further processing, querying, and/or reasoning are performed. Merely ingesting RDF data into data structures that are optimized for query performance (i.e., RDF databases) is a non-trivial task requiring a substantial amount of time. A recent benchmark study using a business intelligence use case showed that most RDF databases struggle to support more than a few billion RDF triples [3], and the product that was able to handle ten billion or more RDF triples – and which could reliably ingest all the data in an automated fashion – took on the order of hours just to load the data (after tweaking for perceived optimal performance). A common optimization performed by RDF databases (and databases in general) during ingest is to assign unique integer identifiers (UIDs) to long byte sequences (e.g., character strings) in order to decrease space consumption and improve join performance (testing equality of UIDs is much faster than for strings). The bijective assignment of UIDs to strings is known as *dictionary encoding*, and since all values in RDF – known as *RDF terms* – are effectively strings of unbounded, finite length, dictionary encoding is particularly important for RDF databases. For this reason, recent research has looked at improving the performance of dictionary-encoding of large RDF datasets by way of parallelism on distributed memory architectures [4], [5]. We have faced the same challenge of dictionary-encoding in implementing our Graph Engine for Multithreaded Systems (GEMS) database [6]. Our solution consists of a parallel RDF parser, a triple table, and a distributed dictionary data structure, exploiting RDF-specific optimizations. With respect to other solutions, our approach employs a Partitioned Global Address Space (PGAS) programming model combined with active messages. Although the specific implementation takes advantage of the underlying runtime library of our specific database, the approach is generalizable to other PGAS libraries with active messages. The GEMS runtime library is named GMT (Global Memory and Threading) [7], and implements a PGAS data model with lightweight software multithreading and data aggregation. These features enhance performance and effectiveness of graph crawling, the basic operation of our RDF database, on commodity clusters. To implement this RDF encoding algorithm we extended GMT, enabling active messages. Herein, we present our solution, characterize its performance, and provide a comparison with previous approaches.

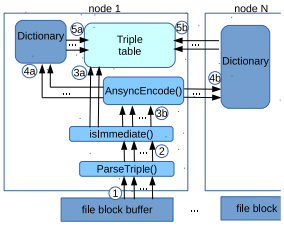
II. Proposed Approach

The GEMS dictionary encoding is implemented as a parallel, lock-free, distributed algorithm resulting in a distributed dictionary data structure and a table of encoded RDF triples. Optionally, these data structures can be saved to disk to avoid repeating the dictionary encoding for the RDF dataset. Once the dataset has been encoded, in-memory data structures can be used to decode UIDs back into RDF terms. The decode operation is typically performed only occasionally during query execution and when presenting results. Similar to [8], the dataset file – provided in the N-Triples format [9] – is divided in blocks and each cluster node encodes a contiguous set of blocks. The main reason to divide the dataset into blocks – rather than loading the whole dataset in memory – is to be able to encode datasets that are larger than the total memory of the whole cluster. A brief description of the procedure follows:

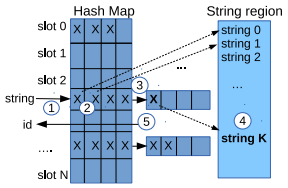
1) Each cluster node reads a file block into a buffer and generates L tasks[1] to perform the encoding

[1]In our evaluation, $L = 1024 * NUM\_WORKERS$ where NUM_WORKERS specifies 15 worker threads per node. See [7] for details.

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.44



(a) Schema of dictionary encoding with GEMS.



(b) Schema of the local dictionary encode.

in parallel. (Each task executes the *EncodeTriples* function described later).

2) While the encoding of a buffer is happening, the node starts to read the next file block into another buffer to overlap disk read time and encoding time.
3) Buffers are swapped and the algorithm repeats from step 1.
4) If the saving mode is enabled, when all file blocks of the node are completed, the node writes the encoded RDF triples into the output file (write phase).

As mentioned, the encoding algorithm uses two main data structures that are distributed across the cluster: the dictionary and the triple table. The dictionary is used to encode each of the three terms of a triple into UIDs. A triple table is a container to store triples where each term has been substituted its corresponding UID (in the form ⟨*subjectUID, predicateUID, objectUID*⟩). UID values are partitioned across the cluster nodes, and an RDF term is assigned a UID by the node such that *nodeid = hash(term) % #nodes*.

Figure 1a is a high-level description of the GEMS dictionary encoding algorithm. At the beginning, each cluster node reads a file block (1). The block is parsed in parallel with L GMT tasks, and each task gets a chunk of the block.[2] The actual algorithm is performed using an optimal number of GMT tasks (1024 per core on our system) to improve the latency hiding of remote communication with software-multithreading. Each task parses a set of triples and, for each triple, recognizes the *subject*, *predicate*, and *object* (2). After the parsing, each task attempts to encode each of the terms as *immediate UIDs*, which means it tries to effectively embed each RDF term directly into a UID. Immediate encoding works only for special cases of RDF terms such as: short string literals; many numeric literals; date/time literals; and common, standard URIs. Our design of immediate UIDs are RDF-specific, taking advantage of knowledge about common RDF terms provided by standards and best practices. The potential time savings using immediate UIDs can be significant, since it allows a task to bypass the dictionary. If a term is encoded as an immediate UID, then it is directly inserted into the triple table (3a); otherwise, the task performs the *AsyncEncode* operation for that term (3b). The *AsyncEncode* consists of applying a string hash function (djb2 hash function) to map the term on a specific cluster node. In case the term belongs to the local node, the local dictionary is used (4a); otherwise, a new task is spawned on the remote node (4b). Note that for each triple, if no terms are encoded as immediate UIDs, three tasks are asynchronously executed to encode the triple's constituent terms. After the local dictionary encoding (4a or 4b), the UID is inserted into the triple table (5a or 5b). Algorithm 1 reports the pseudo-code of the *EncodeTriples* function (corresponding to points 1, 2, and 3 in the schema). The *AsyncEncode* procedure is shown in algorithm 2 and corresponds to point (3a) in the schema. Figure 1b is a schema of the dictionary data structure. The dictionary is composed of: a closed-address, dynamic hash map; and a string region. The hash map is divided into slots (buckets), and each slot is divided into entries. Additional entries may be dynamically linked to each slot, forming a linked list. Also, the dictionary includes a string region that contains the RDF terms that have been encoded. The string region can be expanded by allocating additional regions (not shown in the schema). When an entry is used, it points to the corresponding RDF term in the string region and also contains the assigned UID. Initially, a task identifies the slot based on the hash value of the RDF term (1). Then, the task compares the RDF term to the already-encoded RDF terms pointed to by the entries of the slot (2). When all available entries are used, a new set of entries is allocated (3) and linked to the last set of entries. If the RDF term is found, then the task returns the UID recorded in the corresponding entry; if an empty entry is found, the RDF term is inserted into the string region (4) and a new UID is assigned. Finally, the task returns the UID (5). We found that allocating sets of four entries is a good trade-off to reduce the allocation cost and avoid wasting space. Algorithm 3 reports the pseudo-code of the dictionary encoding procedure described in the schema.

Algorithm 1 EncodeTriples() - encode a set of triples

```
startTriple ← taskChunkStart
while startTriple < taskChunkEnd do
    ParseTriple(start, end, &subject, &predicate, &object)
    AsyncEncode(subject, encodedTriples[i].subject)
    AsyncEncode(predicate, encodedTriples[i].predicate)
    AsyncEncode(object, encodedTriples[i].object)
    startTriple ← GetNextTriple()
end while
gmtWaitExecute()
```

Algorithm 2 AsyncEncode() - encode a string

```
id ← isImmediate(string)
if id ≠ INVALID then
    return id
end if
hashvalue ← hash(string)
slotIdx ← mod(hashvalue, #numSlots)
node ← slotIdx / slotsPerNode
gmtExecuteNB(EncodeLocal(), args)
```

The distribution of RDF terms on the Web typically follows a power-law distribution with a long tail [10]. That is, a few terms are repeated a very large number of times, and most terms are repeated very few times. For this reason, when encoding a term that belongs to a remote node, it is a good strategy to keep track of that operation in a local cache. Additional tasks encoding the same term will first search into the local cache for the term. In case the term is found in the cache, then the task will wait for the (previously established) remote encode to complete in order to obtain the UID. Otherwise, if the term is not in the cache, the task will insert the term in the cache and start the remote encoding procedure. Once a term is in the cache, it will be available until the end of dictionary encoding.

[2] As in [8], we are careful to divide blocks and chunks along triple boundaries in the input file.

Algorithm 3 EncodeLocal() - encode a string in the local dictionary

```
slot ← slots[slotIdx]
while TRUE do
    i ← 0
    while i < SLOT_SIZE do
        entry ← slot [ i ]
        if entry is empty then
            if atomicCAS(entry.string, newString) then
                copyStringInRegion(string)
                entry.id ← assignNewId()
                return entry.id
            end if
        else
            if stringEqual(entry.string, newString) then
                return entry.id
            else
                i ← i + 1
            end if
        end if
    end while
    if nextSlot is not allocated then
        newSlot ← allocate (SLOT_SIZE)
        if not atomicCAS(slot.next , newSlot) then
            free(newSlot)
        end if
    end if
    slot ← slot.next
end while
```

III. Evaluation

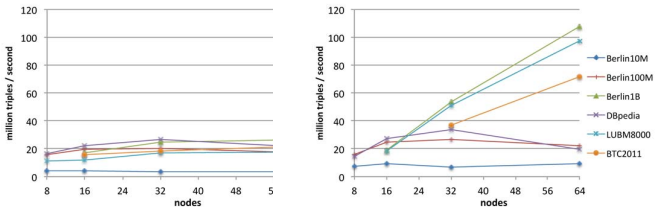
To evaluate our approach, we used two synthetic benchmarks, and several real-world datasets. As synthetic benchmarks we considered the dataset generators from the Berlin benchmark[3] and the Leigh University Benchmark (LUBM) [11]. The Berlin benchmark is intended for comparing RDF database systems in terms of query throughput. We generated datasets containing approximately ten million, 100 million, and one billion triples, referred to herein as Berlin10M, Berlin100M, and Berlin1B, respectively. LUBM is a benchmark for comparing reasoning systems. We generated the dataset for 8,000 universities, referred to herein as LUBM8000. As real-world datasets, we considered the 100 million triple benchmark from DBpedia [12], which consists of RDF data extracted from Wikipedia, the 2011 Billion Triples Challenge (BTC), which consists of crawl of RDF data from the Web, and the RDF version of the Universal Protein Resource (UniProt) [13] dataset, a catalog of proteins. We executed our benchmarks on Pacific Northwest National Laboratory's Olympus supercomputer, listed in the TOP500 [14].

To evaluate the throughput of the algorithm, we run experiments with the dataset entirely loaded in memory before performing the encoding. Note that in this kind of experiment, the maximum dataset we can encode is limited by the global memory size and, hence, some datasets may exceed memory with the given number of cluster nodes. Table I reports the execution times (in seconds) to perform the dictionary encoding with the dataset in memory. In this experiment UniProt exceeded memory in all cluster configurations. Figure 1 reports the throughput (millions triples per second) of the GEMS dictionary encoding without I/O. Results show that even without the bottleneck of the I/O, the algorithm fails to scale linearly when the cache is disabled. In particular it the execution time scales less than linearly up to 32 nodes and in some instances increases with 64 nodes due to the data-skew in the dataset. As expected, execution times with the cache enabled show improved scaling; small datasets like

TABLE I. Execution time in seconds of the dictionary encoding without I/O.

	8-nodes	16-nodes	32-nodes	64-nodes
cache disabled				
Berlin10M	2.28	2.42	2.59	2.82
Berlin100M	6.42	5.05	4.94	5.89
Berlin1B	†	58.38	40.68	36.96
DBpedia	9.30	6.93	5.75	7.43
LUBM8000	97.97	91.56	63.91	61.43
BTC2011	†	139.57	117.84	98.68
cache enabled				
Berlin10M	1.39	1.09	1.47	1.12
Berlin100M	6.23	3.98	3.73	4.44
Berlin1B	†	52.05	18.68	9.27
DBpedia	10.51	5.59	4.56	7.81
LUBM8000	†	59.56	21.64	11.35
BTC2011	†	†	58.87	30.44

† memory exceeded



(a) cache disabled (b) cache enabled

Fig. 1. Dictionary encoding throughput without I/O.

Berlin100M and DBPedia scale up to 16 nodes while larger datasets like Berlin1B, LUBM8000, and BTC2011 scale up to 64 nodes. The local encoding cache mitigates load imbalance by reducing the work of the nodes that "own" high-frequency terms. The distribution of strings when varying the number of nodes can differ significantly. In practice, some nodes will end up encoding a larger number of strings than others. The super-linear scaling of Berlin1B and LUBM8000 with the cache enabled is another example of non-linearity caused by data-skew. In the latter case, the algorithm has slightly higher locality (i.e., it is using the cache more effectively) with 32 nodes than with 16 nodes.

The previous approaches most comparable to ours are a *MapReduce approach* [4] and a *direct MPI approach* [5]. The experiment used for comparison consists in reading the RDF triple files, performing the encoding, and writing the encoded RDF dataset to the filesystem. Both GEMS and MapReduce expect the dataset to be compressed with gzip while MPI expects it to be compressed with LZ0. It is worthwhile to observe that GEMS ingests a single file and also produces a single file. This is not true for the two other solutions: MapReduce ingests and generates multiple output files, and MPI does similarly. The use of multiple files typically improves filesystem performance at the cost of lower user simplicity. The output files produced after encoding have very similar sizes between GEMS and MPI given that the result of the dictionary encoding is the same: a sequence of triples in the form $\langle$*subjectUID, predicateUID, objectUID*$\rangle$ and a set of pairs in the form $\langle$*UID, term*$\rangle$. We executed GEMS experiments with the local encoding cache enabled. Running times for the three RDF dictionary-encoding implementations are shown in

TABLE II. EXECUTION TIME (IN SECONDS) INCLUDING I/O

MapReduce

dataset	8-nodes	16-nodes	32-nodes	64-nodes
Berlin10M	1,131	1,155	1,093	1,156
Berlin100M	1,798	1,544	2,348	1,424
Berlin1B	4,049	2,903	2,462	2,067
DBpedia	952	664	520	474
LUBM8000	4,426	3,434	3,059	3,037
BTC2011	7,036	4,220	3,313	3,061
UniProt	‡	22,515	11,860	7,939

‡ did not finish after 10 hours (36,000 seconds)

MPI

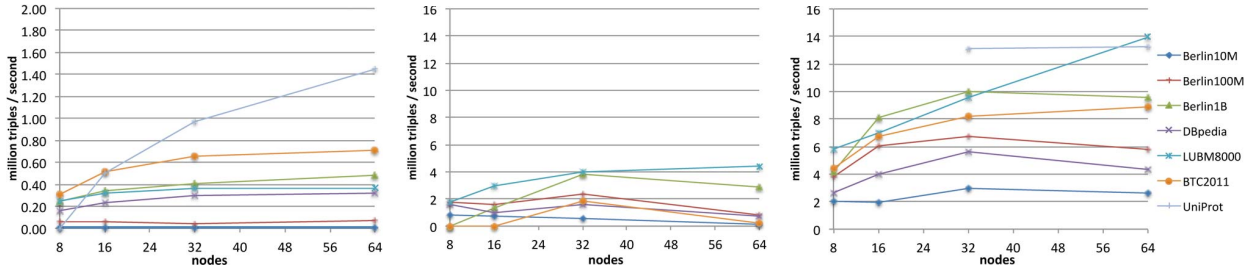
dataset	8-nodes	16-nodes	32-nodes	64-nodes
Berlin10M	12	13	18	102
Berlin100M	57	63	42	128
Berlin1B	†	740	262	350
DBpedia	94	160	94	212
LUBM8000	624	373	278	252
BTC2011	†	†	1,188	8,952
UniProt	†	†	†	†

† memory exceeded

GEMS

dataset	8-nodes	16-nodes	32-nodes	64-nodes
Berlin10M	4.90	5.19	3.37	3.89
Berlin100M	26.19	16.58	14.94	17.22
Berlin1B	237.60	122.67	100.03	104.40
DBpedia	57.76	38.46	27.15	35.52
LUBM8000	189.64	157.36	115.83	79.33
BTC2011	491.15	323.74	266.17	245.48
UniProt	†	†	874.33	863.55

† memory exceeded



(a) MapReduce (b) MPI (c) GEMS

Fig. 2. Dictionary encoding throughput

Table II, and throughput (millions of triples per second) is shown in Figure 2. Note the different scale for the MapReduce throughput compared to the MPI and GEMS throughputs. GEMS is from 1.6 (DBpedia 8 nodes) to 36 (BTC2011 64 nodes) times faster than MPI and from 9.20 (UniProt 64 nodes) to 324.38 (Berlin10M 32 nodes) times faster than MapReduce. This result is somehow expected given the I/O centric design of MapReduce. For small datasets (Berlin10M and Berlin100M), GEMS is over two orders of magnitude faster than MapReduce. The filesystem is a bottleneck for the GEMS approach and the throughput is limited by the read/write operations and the scalability is affected. The MPI implementation also shows very poor scalability with smaller datasets (Berlin10M and Berlin100M) being slower with a higher number of nodes. The reason for the poor scaling of the MPI code is mainly the presence of collectives operations (`MPI_Allreduce`) that limit scalability.

IV. CONCLUSIONS

In this paper, we proposed on approach for high-performance, distributed dictionary-encoding of RDF datasets. This approach is applicable to any system with support for a PGAS programming model and active messages. This particular implementation is part of the GEMS database, and it consists of a parallel RDF parser, a (encoded) triple table, and a distributed dictionary data structure. GEMS is built on top of GMT, a runtime system designed to scale irregular application on commodity clusters. We compared our implementation with two recent approaches known in the literature: an MPI-based implementation and a MapReduce-based implementation (WebPIE). Our results show that the proposed approach is from 9 to 397 times faster than the MapReduce approach, and up to 36 times faster than the MPI approach.

REFERENCES

[1] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 concepts and abstract syntax," W3C, Cambridge, MA, W3C Recommendation, Feb. 2014, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[2] "Linked data — linked data - connect distributed data across the web," http://linkeddata.org/.

[3] P. Boncz and M.-D. Pham, "BSBM V3.1," http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/, Apr. 2013.

[4] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal, "Scalable RDF data compression with mapreduce," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 1, pp. 24–39, 2013.

[5] J. Weaver, *Toward Webscale, Rule-based Inference on the Semantic Web via Data Parallelism*, 2013, ch. A: Dictionary Encoding, pp. 134–144.

[6] A. Morari, V. Castellana, O. Villa, A. Tumeo, J. Weaver, D. Haglin, S. Choudhury, and J. Feo, "Scaling semantic graph databases in size and performance," *Micro, IEEE*, vol. 34, no. 4, pp. 16–26, July 2014.

[7] A. Morari, A. Tumeo, D. Chavarria-Miranda, O. Villa, and M. Valero, "Scaling irregular applications through data aggregation and software multithreading," in *IPDPS '14: IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1126–1135.

[8] J. Weaver and G. T. Williams, "Reducing I/O Load in Parallel RDF Systems via Data Compression," in *HPCSW 2011: 1st Workshop on High-Performance Computing for the Semantic Web*, 2011.

[9] D. Beckett, "RDF 1.1 N-triples," W3C, Cambridge, MA, W3C Recommendation, Feb. 2014, http://www.w3.org/TR/2014/REC-n-triples-20140225/.

[10] L. Ding and T. Finin, "Characterizing the semantic web on the web," in *ISWC 2006: The Semantic Web*. Springer, 2006, pp. 242–257.

[11] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," in *The semantic web*. Springer, 2007, pp. 722–735.

[13] UniProt Consortium *et al.*, "The universal protein resource (uniprot)," *Nucleic acids research*, vol. 36, no. suppl 1, pp. D190–D195, 2008.

[14] "TOP500 - PNNL's Olympus entry," http://www.top500.org/system/177790.