GEMS: Graph database Engine for Multithreaded Systems

Alessandro Morari, Vito Giovanni Castellana, Antonino Tumeo,
Jesse Weaver, David Haglin, Sutanay Choudhury, John Feo
Pacific Northwest National Laboratory
902 Battelle Blvd
Richland, WA 99354, USA
{alessandro.morari, vitoGiovanni.castellana, antonino.tumeo,
jesse.weaver, david.haglin, sutanay.choudhury, john.feo}@pnnl.gov
Oreste Villa
NVIDIA Research
2701 San Tomas Expressway
Santa Clara, CA 95050, USA
ovilla@nvidia.com

***Abstract*—This paper presents GEMS (Graph database Engine for Multithreaded Systems), a software infrastructure that enables large-scale, graph databases on commodity clusters. Unlike current approaches, GEMS implements and explores graph databases by primarily employing graph-based methods. This is reflected in all the layers of the software stack. On one hand, this allows exploiting the space efficiency of graph data structures and the inherent parallelism of some graph algorithms. These features adapt well to the increasing system memory and core counts of modern commodity clusters. On the other hand, however, graph-based methods introduce irregular, fine-grained data accesses with poor spatial and temporal locality, causing performance issues with these systems that are, instead, optimized for regular computation and batched data transfers. Our framework comprises: a SPARQL to data parallel C++ compiler; a library of distributed data structures; and a custom, multithreaded, runtime system. We introduce our stack, motivate its advantages with respect to other solutions, and show how we solved the challenges posed by irregular behaviors. We evaluate our software stack on the Berlin SPARQL benchmark with datasets up to 10 billion graph edges, demonstrating scaling in dataset size and in performance as nodes are added to a cluster.**

I. Introduction

Many fields require organization, management, and analysis of massive amounts of data; such fields include social network analysis, financial risk management, threat detection in complex network systems, and medical and biomedical databases. These are all examples of big data analytics in which dataset sizes increase exponentially. These application fields pose operational challenges not only in terms of sheer size but also in terms of time to solution because quickly answering queries is essential to obtain market advantages, avoid critical security issues, or prevent life threatening health problems.

Semantic graph databases seem a promising solution to store, manage, and query the large and heterogeneous datasets of these application fields. Such datasets present an abundance of relations among many elements. Semantic graph databases organize the data in the form of subject-predicate-object triples following the Resource Description Framework (RDF) data model. A set of triples naturally represent a labeled, directed multigraph. An analyst can query semantic graph databases through languages such as SPARQL in which the fundamental query operation is graph matching. This is different from conventional relational databases that employ schema-specific tables to store data and perform select and conventional join operations when executing queries. With relational approaches, graph-oriented queries on large datasets can quickly become unmanageable in both space and time due to the large sizes of immediate results created when performing conventional joins.

Graphs are memory efficient data structures for storing data that is heterogeneous or not itself rigidly structured. Graph methods based on edge traversal are inherently parallel because the system can potentially generate parallel activities for (single or group of) vertex or edge to be traversed. Modern commodity clusters – composed of nodes with increasingly higher core counts, larger main memory, and faster network interconnections – are an interesting target platform for in-memory crawling of big graphs, potentially enabling scaling in size by adding more nodes, while maintaining constant throughput. However, graph-based methods are irregular: they exhibit poor spatial and temporal locality, perform fine-grained data accesses, usually present high synchronization intensity, and have datasets with high data skew which can lead to severe load imbalance. These characteristics make the execution of graph exploration algorithms on commodity clusters challenging. In fact, modern clusters are optimized for applications that exhibit regular behaviors, high (floating point) arithmetic intensity, and have easily partition able datasets with high locality: they integrate multicore processors with high flop ratings and complex cache hierarchies, and networks that reach peak bandwidth only with large, batched data transfers. However, the parallelism of graph-based methods can be exploited to realize multithreaded execution models that create and manage an oversubscription of tasks to cores, allowing for toleration of data access latency, rather than reducing it through locality.

In this paper, we present GEMS (Graph database Engine for Multithreaded Systems), a complete software stack that

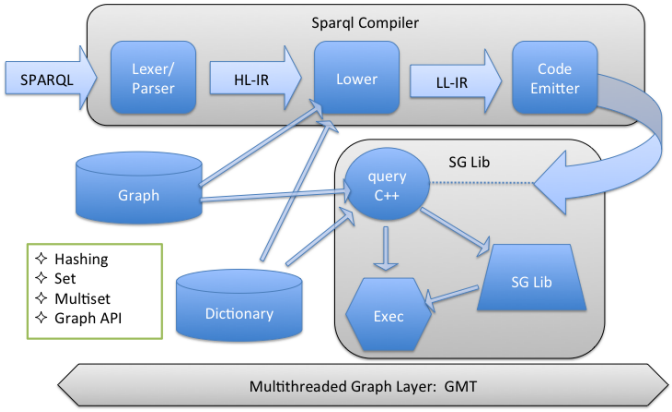


Fig. 1: Architecture of GEMS - Graph database Engine for Multithreaded Systems.

implements a semantic graph database for big data analytics on commodity clusters. Currently available semantic graph databases usually implement mechanisms to store, retrieve, and query triples on top of conventional relational databases, or still resort to relational approaches for some of their components. In contrast, GEMS approaches semantic graph databases primarily with graph-based methods at all the levels of the stack. GEMS includes: a compiler that converts SPARQL queries to data parallel graph pattern matching operations in C++; a library of parallel algorithms and related, distributed data structures; and a custom, multithreaded, runtime layer for commodity clusters.

We describe GEMS and its underlining runtime, discussing how we solved limitations of commodity clusters with applications that exhibit irregular behaviors to enable scaling in size and performance.

II. GEMS Overview

Figure 1 provides an overview of the architecture of our Graph database Engine for Multithreaded Systems (GEMS). As previously introduced, GEMS comprises: a SPARQL to C++ compiler; a Semantic Graph library (SGLib) of supporting data structures such as graph and dictionary with the related parallel Application Programming Interface (API) to access them; and a Global Memory and Threading (GMT) runtime layer.

The top layer consists of the compilation phases. The compiler transforms the input SPARQL queries into intermediate representations that are analyzed for optimization opportunities. Potential optimization opportunities are discovered at multiple levels. Depending on statistics of the datasets, certain query clauses can be moved, enabling early pruning of the search. Then, the optimized intermediate representation is converted into C++ code that contains calls to the SGLib API. SGLib APIs completely hide the low-level APIs of GMT, exposing to the compiler a lean, simple, pseudo-sequential shared-memory programming model. SGLib manages the graph database and query execution. SGLib generates the graph database and the related dictionary by ingesting the triples. Triples can, for example, be RDF triples stored in N-Triples format. The approach implemented by our system to extract information from the semantic graph database is to solve a structural graph pattern matching problem. GEMS employs a variation of Ullmann's subgraph isomorphism algorithm [1]. The GMT layer provides the key features that enable management of the data structures and load balancing across the nodes of the cluster. GMT is co-designed with the upper layers of the graph database engine so as to better support the irregularity of graph pattern matching operations. GMT provides a Partitioned Global Address Space (PGAS) data model, hiding the distributed nature of the cluster. GMT exposes to SGLib an API that permits allocating, accessing, and freeing data in the virtual shared memory. Differently from other PGAS libraries, GMT employs a control model typical of shared-memory systems: fork-join parallel constructs that generate thousands of lightweight tasks. These lightweight tasks allow hiding the latency for accessing data on remote cluster nodes; they are switched in and out of processor cores while communication proceeds. Finally, GMT aggregates operations before communicating to other nodes, increasing network bandwidth utilization.

Figure 2 shows an example RDF dataset and a related query in different stages of compilation. Figure 2a shows the dataset in N-Triples format (a common serialization format for RDF), and Figure 2b shows the corresponding graph representation. Figure 2c shows the SPARQL description of the query, Figure 2d illustrates its graph pattern, and Figure 2e shows the pseudocode generated by the compiler and executed by GMT through SGLib.

GEMS has minimal system-level library requirements: besides Pthreads, it only needs MPI for the GMT communication layer and Python for some compiler phases and for glue scripts. Currently, GEMS also requires x86-compatible processors because GMT employs optimized context switching routines. However, this requirement may be removed by developing specific context switching routines for other architectures.

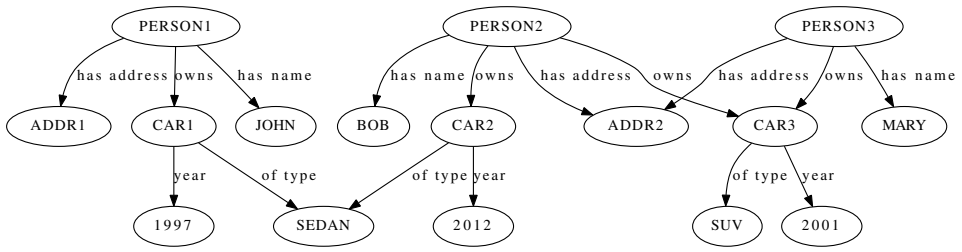
III. Related approaches

Many commercial and open source SPARQL engines are available. We can distinguish between purpose-built databases for the storage and retrieval of triples (triplestores), and solutions that try to map triple stores on top of existing commercial databases, usually relational SQL-based systems. However, obtaining feature-complete SPARQL-to-SQL translation is difficult, and may introduce performance penalties. Translating SPARQL to SQL implies the use of relational algebra to perform optimizations, and the use of classical relational operators (e.g., conventional joins and selects) to execute the query. By translating SPARQL to graph pattern matching operations, GEMS reduces the overhead for intermediate data structures and can exploit optimizations that look at the execution plan (i.e., order of execution) from a graph perspective.

SPARQL engines can further be distinguished between solutions that process queries in-memory and solutions that store data on disks and perform swapping. Jena (with the ARQ SPARQL engine [2]), Sesame [3], and Redland [4] (aka librdf) are all example of RDF libraries that natively implement in-memory RDF storage and support integration with some disk-based, SQL backends. OpenLink Virtuoso [5] implements a RDF/SPARQL layer on top of their SQL-based column store for which multi-node, cluster support is available. GEMS

```
PERSON1 has_name    JOHN  .
PERSON1 has_address ADDR1 .
PERSON1 owns        CAR1  .
CAR1    of_type     SEDAN .
CAR1    year        1997  .
PERSON2 has_name    BOB   .
PERSON2 has_address ADDR2 .
PERSON2 owns        CAR2  .
CAR2    of_type     SEDAN .
CAR2    year        2012  .
PERSON2 owns        CAR3  .
CAR3    of_type     SUV   .
CAR3    year        2001  .
PERSON3 has_name    MARY  .
PERSON3 has_address ADDR2 .
PERSON3 owns        CAR3  .
```

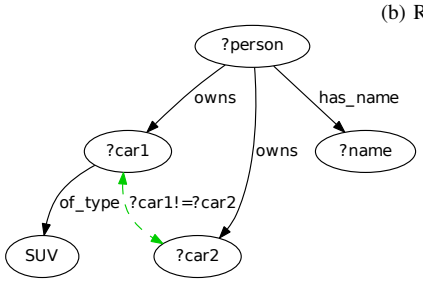
(a) Dataset in simplified N-Triples format



(b) RDF Graph

```
SELECT DISTINCT ?name
WHERE {
   ?person owns     ?car1 .
   ?person owns     ?car2 .
   ?person has_name ?name .
   ?car1   of_type  SUV   .
   FILTER(?car1 != ?car2)
}
```

(c) Simplified SPARQL query



(d) Pattern graph

```
1  has_name = get_label("has_name")
2  of_type  = get_label("of_type")
3  owns     = get_label("owns")
4  suv      = get_label("SUV")
5  forall e1 in edges(*, of_type, suv)
6    ?car1 = source_node(e1)
7    forall e2 in edges(*, owns, ?car1)
8      ?person = source_node(e2)
9      forall e3 in edges(?person, owns, *)
10       ?car2 = target_node(e3)
11       if (?car1 != ?car2)
12         forall e4 in edges(?person,has_name,*)
13           ?name = target_node(e4)
14           tuples.add(<?name>)
15 distinct(tuples)
```

(e) Pseudocode

Fig. 2: Example RDF Dataset and example query: *"return the names of all persons owning at least two cars, of which at least one is a SUV"*.

adopts in-memory processing: it stores all data structures in RAM. In-memory processing potentially allows increasing the dataset size while maintaining constant query throughput by adding more cluster nodes.

Some approaches leverage MapReduce infrastructures for RDF-encoded databases. SHARD [6] is a triplestore bu ilt on top of Hadoop, while YARS2 [7] is a bulk-synchronous, distributed, query answering system. Both exploit hash partitioning to distribute triples across nodes. These approaches work well for simple index lookups, but they also present high communication overheads for moving data through the network with more complex queries, as well as introduce load-balancing issues in the pr esence of data skew. More general graph libraries, such as Pregel [8], Giraph [9], and GraphLab [10] may also be exploited to explore semantic databases, once the source data have been converted into a graph. However, they require significant additions to work in a database environment, and they still rely on bulk-synchronous, parallel models that do not perform well for large and complex queries. Our system relies on a custom runtime that provides specific features to support exploration of a semantic database through graph-based methods.

Urika is a commercial shared memory system from YarcData [11] targeted at big data analytics. Urika exploits custom nodes with purpose-built multithreaded processors (barrel processors with up to 128 threads and a very simple cache) derived from the Cray XMT. Beside multithreading, which allows tolerating latencies for accessing data on remote nodes, the system has hardware support for a scrambled global address space and fine-grained synchronization. These features allow more efficient execution of irregular applications, such as graph exploration. On top of this hardware, YarcData interfaces with the Jena framework to provide a front-end API. GEMS, instead, exploits clusters built with commodity hardware that are cheaper to acquire and maintain, and which are able to evolve more rapidly than custom hardware.

IV. GMT: Addressing Limitations of Commodity Clusters with Irregular Algorithms

The GMT runtime system enables GEMS to scale in size and performance on commodity clusters. GMT is built around three main "pillars": *global address space*, latency tolerance through fine-grained software *multithreading*, and remote data access *aggregation* (also known as coalescing). Global address space (through PGAS) relieves the other layers of GEMS from partitioning the data structures and from orchestrating communication. Message aggregation maximizes network bandwidth utilization, despite the small data accesses typical of graph methods on shared-memory systems. Fine-grained multithreading allows hiding the latency for remote data transfers, and the added latency for aggregation, by exploiting the inherent parallelism of graph algorithms. Figure 3a shows the high-level design of GMT. Each node executes an instance of GMT. Different instances communicate through *commands*, which describe data, synchronization, and thread management operations. GMT is a parallel runtime with three types of specialized threads. The main idea is to exploit the cores of modern processors to support the functionalities of the runtime. The specialized threads are:

Worker: executes application code, in the form of lightweight user tasks, and generates commands directed to other nodes;

Helper: manages global address space and synchronization, and handles incoming commands from other nodes;

Communication Server: endpoint for the network, it manages all incoming/outgoing communication at the node level in form of network messages, which contain the commands.

The specialized threads are implemented as *POSIX* threads, each one pinned to a core. The communication server employs MPI to send and receive messages to and from other nodes.

There are multiple helpers and workers per node (usually an equal number, although this is one of the tunable parameters depending on the target machine) and a single communication server.

SGLib contains data structures that are implemented using shared arrays in GMT's virtual global address space. Among them, there are the graph data structure and the related dictionary. The dictionary is used to map vertex and edge labels (actually RDF terms) to unique integer identifiers. This allows us to compress the graph representation in memory as well as perform label/term comparisons much more efficiently. Dictionary encoding is common practice in database systems. The SPARQL-to-C++ compiler assumes to operate on a shared-memory system and does not need to reason about the physical partitioning of the database. However, as is common in PGAS libraries, GMT also exposes locality information, allowing reduction of data movements whenever possible. Because graph exploration algorithms mostly have loops that run through edge or vertex lists, GMT provides a parallel loop construct that maps loop iterations to lightweight tasks. GMT supports task generation from nested loops and allows specifying the number of iterations of a loop mapped to a task. GMT also allows controlling code locality, enabling to spawn (or move) tasks on preselected nodes, instead of moving data. SGLib routines exploit these features to better manage its internal data structures. SGLib routines access data through *put* and *get* communication primitives, moving them into local space for manipulation and writing them back to the global space. The communication primitives are available both with blocking and non-blocking semantics. GMT also provides atomic operations, such as atomic addition and test-and-set, on data allocated in the global address space. SGLib exploits them to protect parallel operations on the graph datasets and to implement global synchronization constructs for database management and querying.

A. Aggregation

Graph exploration algorithms present fine grained data accesses: for-loops effectively run through edges and/or vertices represented by pointers, and each pointer may point to a location in a completely unrelated memory area. With partitioned datasets on distributed memory systems, expert programmers have to implement by hand optimizations to aggregate requests and reduce the overhead due to small network transactions. GMT hides these complexities from the other layers of GEMS by implementing automatic message aggregation.

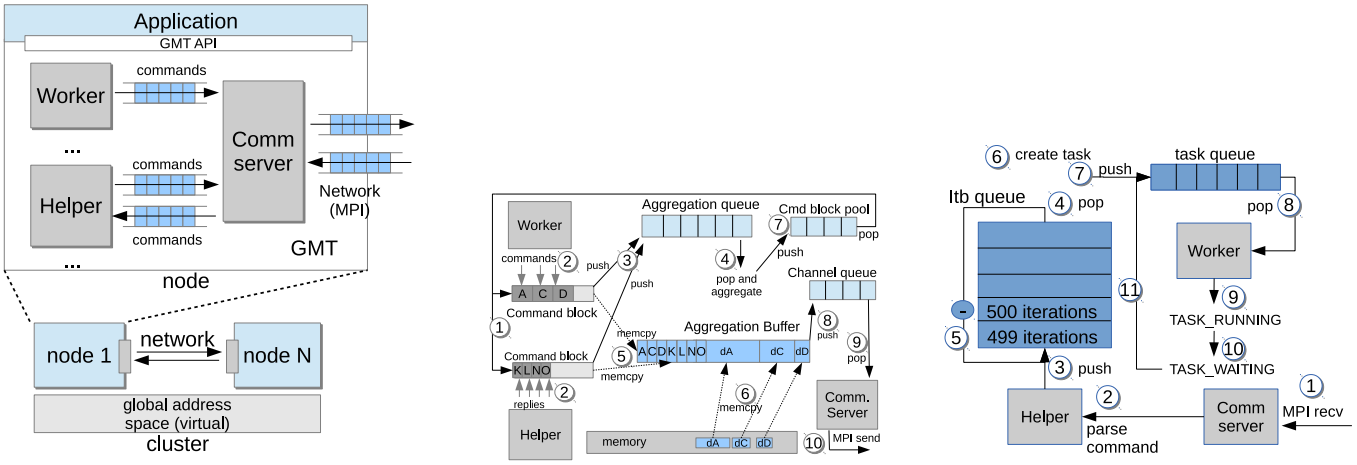
GMT collects commands directed towards the same destination nodes in aggregation queues. GMT copies commands and their related data (e.g., values requested from the global address space with a get) into aggregation buffers, and sends them in bulk. Commands are then unpacked and executed at the destination node. At the node level, GMT employs high-throughput, non-blocking aggregation queues, which support concurrent access from multiple workers and helpers. Accessing these queues for every generated command would have a very high cost. Thus, GMT employs a two-level aggregation mechanism: workers (or helpers) initially collect commands in local command blocks, and then they insert command blocks into the aggregation queues. Figure 3b describes the aggregation mechanism. When aggregation starts, workers (or helpers) request a pre-allocated command block from the command block pool (1). Command blocks are pre-allocated and reused for performance reasons. Commands generated during program execution are collected into the local command block (2). A command block is pushed into aggregation queues when: (a) it is full, or (b) when it has been waiting longer than a pre-determined time interval. Condition (a) is true when all the available entries are occupied with commands, or when the equivalent size in bytes of the commands (including any attached data) reaches the size of the aggregation buffer. Condition (b) allows setting a (configurable) upper bound for the latency added by aggregation. After pushing a command block, when a worker or a helper finds that the aggregation queue has sufficient data to fill an aggregation buffer, it starts popping command blocks from the aggregation queue and copying them with the related data into an aggregation buffer (4, 5, and 6). Aggregation buffers also are pre-allocated and recycled to save memory space and eliminate allocation overhead. After the copy, command blocks are returned to the command block pool (7). When the aggregation buffer is full, the worker (or helper) pushes it into a channel queue (8). Channel queues are high-throughput, single-producer, single-consumer queues that workers and helpers use to exchange data with the communication server. If the communication server finds a new aggregation buffer in one of the channel queues, it pops it (9) and performs a non-blocking MPI send (10). The aggregation buffer is then returned into the pool of free aggregation buffers.

The size of aggregation buffers and the time intervals for pushing out aggregated data are configurable parameters that depends on the interconnection of the cluster on which GEMS resides. Buffers should be sufficiently large to maximize network throughput, while time intervals should not increase the latency over the values maskable through multithreading.

B. Multithreading

Concurrency, through fine-grained software multithreading, allows tolerance for both the latency for accessing data on remote nodes and the added latency for aggregating communication operations. Each worker executes a set of GMT tasks. The worker switches among tasks' contexts every time it generates a blocking command that requires a remote memory operation. The task that generated the command executes again only when the command itself completes (i.e., it gets a reply back from the remote node). In case of non-blocking commands, the task continues executing until it encounters a wait primitive.

GMT implements custom context switching primitives that avoid some of the lengthy operations (e.g., saving and restoring signal mask) performed by the standard *libc* context switching routines. Figure 3c schematically shows how GMT executes a task. A node receives a message containing a *spawn* command (1) that is generated by a worker on a remote node when encountering a parallel construct. The communication server passes the buffer containing the command to a helper that parses the buffer and executes the command (2). The helper then creates an *iteration block* (itb). The itb is a data structure that contains the function to execute, the arguments of the function itself, and a counter of tasks that execute the same function. This way of representing a set of tasks avoids the cost



(a) Architecture of GMT (b) Aggregation mechanism (c) Fine-grained multithreading management

Fig. 3: GMT architecture, aggregation mechanism and multithreading

of creating a large number of function arguments and sending them over the network. In the following step, the helper pushes the iteration block into the itb queue (3). Then, an idle worker pops an itb from the itb queue (5), decreases the counter of t and pushes it back into the queue (6). The worker creates t tasks (6) and pushes them into its private task queue (7). At this point, the idle worker can pop a task from its task queue (8). If the task is executable (i.e., all the remote operations completed), the worker restores the task's context and executes it (9). Otherwise, it pushes the task back into the task queue. If the task contains a blocking remote request, the task enters a waiting state (10) and is reinserted into the task queue for future execution (11).

This mechanism provides load balancing at the node level because each worker gets new tasks from the itb queue as soon as its task queue is empty. At the cluster level, GMT evenly splits tasks across nodes when it encounters a parallel for-loop construct. If, instead, a task creation addresses a specific node, the task is created on that node.

V. Experimental results

We evaluated GEMS on the Olympus supercomputer at Pacific Northwest National Laboratory's Institutional Computing center, listed in the TOP500 [12]. Olympus is a cluster of 604 nodes interconnected through a QDR Infiniband switch with 648 ports (theoretical peak of 4GB/s). Each of Olympus' node features two AMD Opteron 6,272 processors at 2.1 GHz and 64 GB of DDR3 memory clocked at 1,600 MHz. Each socket hosts eight processor modules (two integer cores, one floating point core per module) on two different dies, for a total of 32 integer cores per node. We configured the GEMS stack with 15 workers, 15 helpers, and 1 communication server per node. Each worker hosts up to 1,024 lightweight tasks. We measured the MPI bandwidth of Olympus with the OSU Micro-Benchmarks 3.9 [13], reaching a peak (around 2.8 GB/s) with messages of at least 64 KB. Therefore, we set the aggregation buffer size at 64 KB. Each communication channel hosts up to four buffers. There are two channels per helper and one channel per worker. We initially present some synthetic benchmarks of the runtime, highlighting the combined effects of multithreading and aggregation to maximize network bandwidth utilization. We then show experimental results of the whole GEMS infrastructure on a well established benchmark, the Berlin SPARQL Benchmark (BSBM) [14].

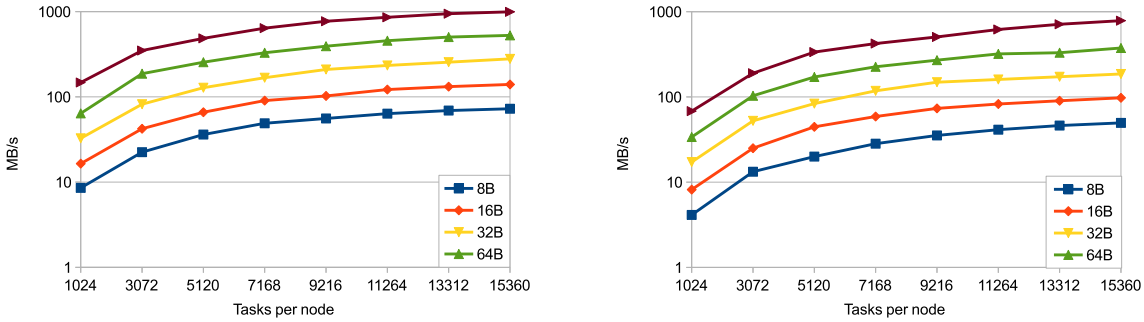
A. Synthetic Benchmarks

Figure 4 shows the transfer rates reached by GMT with small messages (from 8 to 128 bytes) when increasing the number of tasks. Every task executes 4,096 blocking put operations. Figure 4a shows the bandwidth between two nodes, and Figure 4b shows the bandwidth among 128 nodes. The figures show how increasing the concurrency increases the transfer rates, because there is a higher number of messages that GMT can aggregate. For example, across two nodes (Figure 4a) with 1,024 tasks each, puts of eight bytes reach a bandwidth of 8.55 MB/s. With 15,360 tasks, instead, GMT reaches 72.48 MB/s. When increasing message sizes to 128 bytes, 15,360 tasks provide almost 1 GB/s. For reference, 32 MPI processes with 128B messages only reach $72.26 MB/s$. With more destination nodes, the probability of aggregating enough data to fill a buffer for a specific remote node decreases. Although there is a slight degradation, Figure 4b shows that GMT is still very effective. For example, 15,360 tasks with 16B messages reach 139.78 MB/s, while 32 MPI processes only provide up to 9.63 MB/s.

B. GEMS

BSBM defines a set of SPARQL queries and datasets to evaluate the performance of semantic graph databases and systems that map RDF into other kinds of storage systems. Berlin datasets are based on an e-commerce use case with millions to billions of commercial transactions, involving many product types, producers, vendors, offers, and reviews. We run queries one through six of the Business Intelligence use-case on datasets with 100M, 1B, and 10B triples.

The subtables of Table I respectively show the build time of the database and the execution time of the queries on 100M (Subtable Ia), 1B (Subtable Ib), and 10B (Subtable Ic) triples, while progressively increasing the number of cluster nodes. Sizes of the input files respectively are 21 GB (100M), 206



(a) Transfer rates of put operations between 2 nodes while increasing concurrency

(b) Transfer rates of put operations among 128 nodes (one to all) while increasing concurrency

Fig. 4: Synthetic benchmarks showing aggregation and multithreading effects

Nodes	2	4	8	16
build	199.00	106.99	59.85	33.42
Q1	1.83	1.12	0.67	0.40
Q2	0.07	0.07	0.07	0.05
Q3	4.07	2.73	1.17	0.65
Q4	0.13	0.13	0.14	0.15
Q5	0.07	0.07	0.07	0.11
Q6	0.01	0.02	0.02	0.03

(a) 100M triples, 2 to 16 nodes

Nodes	8	16	32	64
build	628.87	350.74	200.54	136.69
Q1	5.65	3.09	1.93	2.32
Q2	0.30	0.34	0.23	0.35
Q3	12.79	6.88	4.50	2.76
Q4	0.31	0.25	0.22	0.27
Q5	0.11	0.12	0.14	0.18
Q6	0.02	0.03	0.04	0.05

(b) 1B triples, 8 to 64 nodes

Nodes	64	128
build	1066.27	806.55
Q1	27.14	39.78
Q2	1.48	1.91
Q3	24.27	18.32
Q4	2.33	2.91
Q5	2.13	2.82
Q6	0.40	0.54

(c) 10B triples, 64 and 128 nodes

TABLE I: Time (in seconds) to build the database and execute BSBM queries 1-6 with 100M, 1B and 10B triples.

GB (1B), and 2 TB (10B). In all cases, the build time scales with the number of nodes. Considering all the three subtables together, we can appreciate how GEMS scales in dataset sizes by adding new nodes, and how it can exploit the additional parallelism available. With 100M triples, Q1 and Q3 scale for all the experiments up to 16 nodes. Increasing the number of nodes for the other queries, instead, provides constant or slightly worse execution time. Their execution time is very short (under 0.5 seconds), and the small dataset does not provide sufficient data parallelism. These queries only have two graph walks with two-level nesting and, even with larger datasets, GEMS is able to exploit all the available parallelism already with a limited number of nodes. Furthermore, the database has the same overall size, but is partitioned on more nodes, thus the communication increases, slightly reducing the performance. With 1B triples, we see similar behavior. In this case, however, Q1 stops scaling at 32 nodes. With 64 nodes, GEMS can execute queries on 10B triples. Q3 still scales in performance up to 128 nodes, while the other queries, except Q1, approximately maintain stable performance. Q1 experiences the highest decrease in performance when using 128 nodes because its tasks present higher communication intensity than the other queries, and GEMS already exploited all the available parallelism with 64 nodes. These data confirm that GEMS can maintain constant throughput when running sets of mixed queries in parallel, i.e., in typical database usage.

VI. Conclusions

In this paper we presented GEMS, a full software stack for semantic graph databases on commodity clusters. Different from other solutions, GEMS proposes an integrated approach that primarily utilizes graph-based methods across all the layers of its stack. GEMS includes a SPARQL-to-C++ compiler, a library of algorithms and data structures, and a custom runtime. The custom runtime (GMT - Global Memory and Threading) provides to all the other layers several features that simplify the implementation of the exploration methods and makes more efficient their execution on commodity clusters. GMT provides a global address space, fine-grained multithreading (to tolerate latencies for accessing data on remote nodes), remote message aggregation (to maximize network bandwidth utilization), and load balancing. We have demonstrated how this integrated approach provides scaling in size and performance as more nodes are added to the cluster.

References

[1] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.
[2] "ARQ - A SPARQL Processor for Jena." [Online]. Available: http://jena.sourceforge.net/ARQ/
[3] "openRDF.org, home of Sesame." [Online]. Available: http://www.openrdf.org
[4] "Redland RDF Libraries." [Online]. Available: http://librdf.org
[5] "Virtuoso Universal Server." [Online]. Available: http://virtuoso.openlinksw.com
[6] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store," in *PSI EtA '10: Programming Support Innovations for Emerging Distributed Applications*, 2010, pp. 4:1–4:5.
[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "YARS2: a federated repository for querying graph structured data from the web," in *ISWC'07/ASWC'07: 6th International Semantic Web and 2nd Asian Semantic Web Conference*, 2007, pp. 211–224.
[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD '10: ACM International Conference on Management of data*, 2010, pp. 135–146.
[9] "Apache Giraph." [Online]. Available: http://incubator.apache.org/giraph/
[10] "Graphlab." [Online]. Available: http://graphlab.org
[11] "YarcData, Inc. Urika Big Data Graph Appliance." [Online]. Available: http://www.cray.com/Products/ BigData/uRiKA.aspx
[12] "TOP500 - PNNL's Olympus entry." [Online]. Available: http://www.top500.org/system/177790
[13] "OSU Micro-Benchmarks." [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/
[14] C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 1–24, 2009.