Scaling Semantic Graph Databases in Size and Performance

Alessandro Morari
Vito Giovanni Castellana
Pacific Northwest National Laboratory

Oreste Villa
Nvidia Research

Antonino Tumeo
Jesse Weaver
David Haglin
Sutanay Choudhury
John Feo
Pacific Northwest National Laboratory

GEMS is a full software system that implements a large-scale, semantic graph database on commodity clusters. Its framework comprises a SPARQL-to-C++ compiler, a library of distributed data structures, and a custom multithreaded runtime library. The authors evaluated their software stack on the Berlin SPARQL benchmark with datasets of up to 10 billion graph edges, demonstrating scaling in dataset size and performance as they added cluster nodes.

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 time to solution, because quickly answering queries is essential to obtaining market advantages, avoiding critical security issues, or preventing life-threatening health problems.

Semantic graph databases seem to be a promising solution for storing, managing, and querying 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 represents 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 differs 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, owing to the large sizes of intermediate results created when conventional joins are performed.

Graphs are memory-efficient data structures for storing data that is heterogeneous or not rigidly structured. Graph methods based on edge traversal are inherently parallel

Published by the IEEE Computer Society

Related Work in Graph Databases

Currently, many commercial and open source SPARQL engines are available. We can distinguish between purpose-built databases for the storage and retrieval of triples (*triple stores*), 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 could introduce performance penalties. Translating SPARQL to SQL implies the use of relational algebra to perform optimizations and the use of classical relational operators (such as 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 (that is, the execution order) from a graph perspective.

SPARQL engines can be further distinguished between solutions that process queries in memory and solutions that store data on disks and perform swapping. Jena (with the ARQ SPARQL engine [http://jena.sourceforge.net/ARQ]), Sesame (www.openrdf.org), and Redland (also called librdf [http://librdf.org]) are all example of RDF libraries that natively implement in-memory RDF storage and support integration with some disk-based, SQL back ends. OpenLink Virtuoso (http://virtuoso.openlinksw.com) implements an RDF/SPARQL layer on top of its SQL-based column store, for which multinode cluster support is available. GEMS adopts in-memory processing, storing 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 is a triple store built on top of Hadoop,[1] whereas YARS2 is a bulk-synchronous, distributed, query-answering system.[2] Both Shard and YARS2 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, and they introduce load-balancing issues in the presence of data skew.

More general graph libraries—such as Pregel,[3] Giraph (http://incubator.apache.org/giraph), and GraphLab (http://graphlab.org)—can 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 (www.yarcdata.com/Products) 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. Besides 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. SGEM, instead, exploits clusters built with commodity hardware that are cheaper to acquire and maintain, and which can evolve more rapidly than custom hardware.

References

1. K. Rohloff and R.E. Schantz, "High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store," *Proc. Programming Support Innovations for Emerging Distributed Applications*, 2010, pp. 4:1-4:5.
2. A. Harth et al., "YARS2: A Federated Repository for Querying Graph Structured Data from the Web," *Proc. 6th Int'l Semantic Web and 2nd Asian Semantic Web Conf.*, 2007, pp. 211-224.
3. G. Malewicz et al., "Pregel: A System for Large-Scale Graph Processing," *Proc. ACM Int'l Conf. Management of Data*, 2010, pp. 135-146.

because the system can potentially generate a task for each (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 (so-called data scaling[1]) 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, their processors implement deep and complex cache hierarchies optimized for locality and regular computation, and their networks reach peak bandwidth only with large, batched data transfers. However, the

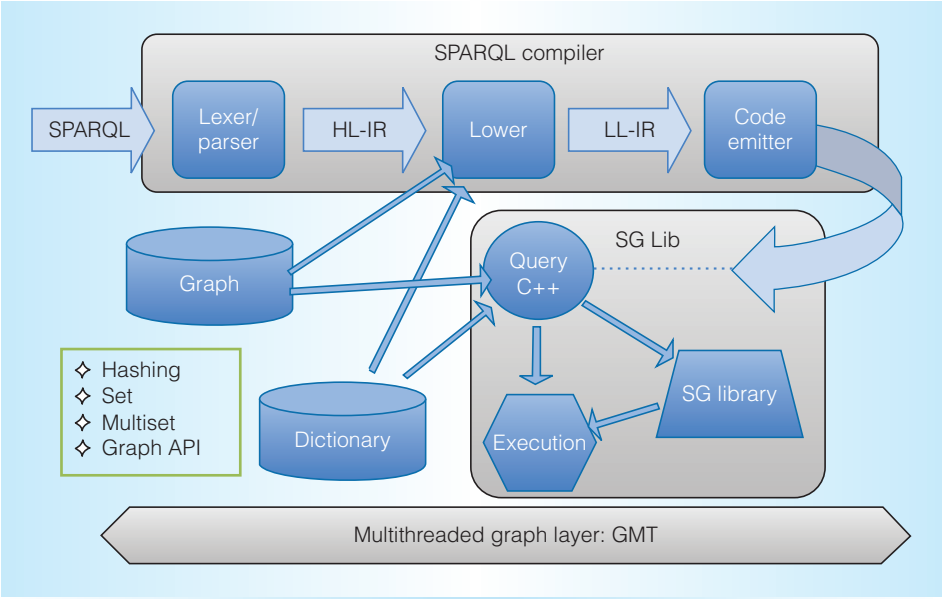


Figure 1. Graph Engine for Multithreaded Systems (GEMS) architecture. GEMS comprises a SPARQL-to-C++ compiler, a Semantic Graph library (SGLib) of supporting data structures, and a Global Memory and Threading (GMT) runtime layer.

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 article, we present GEMS (Graph Engine for Multithreaded Systems), a full software stack that implements a semantic graph database for big data analytics on commodity clusters. Currently available 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 components (see the "Related Work in Graph Databases" sidebar for more information). In contrast, GEMS implements a semantic graph database primarily with graph-based algorithms 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.

In designing SGEM, we were able to work around the limitations of commodity with irregular applications (that is, large-scale graph crawling) by combining lightweight software multithreading with a partitioned global address space and network messages aggregation, enabling scaling in size and performance of graph databases.

GEMS overview

Figure 1 provides an overview of the GEMS architecture. GEMS comprises a SPARQL-to-C++ compiler, a Semantic Graph library (SGLib) of supporting data structures such as the graph and dictionary with the related parallel 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 the datasets' statistics, 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, pseudosequential shared-memory programming model. SGLib manages the

graph database and query execution, and generates the graph database and the related dictionary by ingesting the triples. Triples can, for example, be RDF triples stored in the N-Triples format, a common serialization format for the RDF.

Our system's approach to extracting 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.[2] The GMT layer provides the key features that enable management of the data structures and load balancing across the cluster's nodes. GMT is codesigned with the upper layers of the graph database engine to better support the irregularity of graph pattern-matching operations. GMT provides a partitioned global address space (PGAS) data model, hiding the complexity of the distributed memory system. GMT exposes to SGLib an API that permits allocating, accessing, and freeing data in the global address space. Unlike 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 the N-Triples format, 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 needs only a library supporting the message passing interface (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, specific context-switching routines can be developed for other architectures to avoid this requirement.

GMT: Addressing commodity clusters' limitations

The GMT runtime system enables GEMS to scale in size and performance on commodity clusters. GMT is built on three main pillars: global address space, latency tolerance through fine-grained software multithreading, and remote-data-access aggregation (also called *coalescing*).

The global address space (through PGAS) relieves the other layers of GEMS from partitioning the data structures and 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, as well as the additional latency for aggregation, by exploiting the inherent parallelism of graph algorithms.

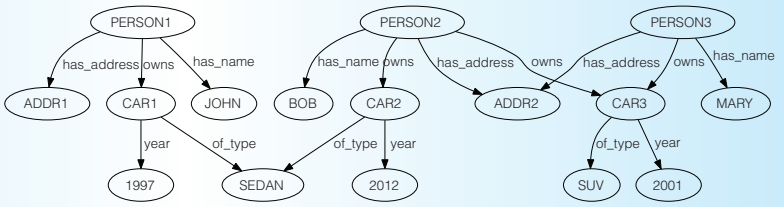
Figure 3a shows the high-level design of GMT. Each node executes an instance of GMT. Different instances communicate through *commands,* which describe data, synchronization, and thread management operations. GMT is a parallel runtime library with three types of specialized threads. The main idea is to exploit modern processor cores to support the runtime library's functionalities. The specialized threads are

- *the worker,* which executes application code in the form of lightweight user tasks and generates commands directed to other nodes;
- *the helper,* which manages global address space and synchronization, and handles incoming commands from other nodes; and
- *the communication server,* the endpoint for the network, which manages all incoming and outgoing communication at the node level in the form of network messages that contain the commands.

The specialized threads are implemented as Posix threads, each pinned to a core. The communication server employs an MPI to

```
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   .
```

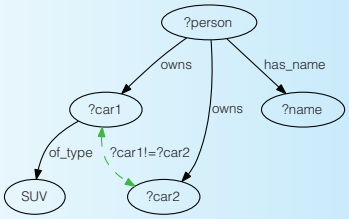
(a)



(b)

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

(c)



(d)

```
has_name = get_label("has_name")
of_type  = get_label("of_type")
owns     = get_label("owns")
suv      = get_label("SUV")
forall e1 in edges(*, of_type, suv)
       ?car1 = source_node(e1)·
       forall e2 in edges(*, owns, ?car1)
         ?person = source_node(e2)·
         forall e3 in edges(?person, owns, *)
           ?car2 = target_node(e3)
           if (?car1 != ?car2)
             forall e4 in edges(?person,has_name,*)
               ?name = target_node(e4)
               tuples.add(<?name>)
distinct(tuples)
```

(e)

Figure 2. Example Resource Description Framework (RDF) dataset and related query, "return the names of all persons owning at least two cars, of which at least one is an SUV." Dataset in simplified N-Triples format (a), RDF graph (b), simplified SPARQL query (c), pattern graph (d), and pseudocode (e).

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 that depend on the target machine) and a single communication server.

SGLib contains data structures that are implemented using shared arrays in GMT's global address space. Among them are the graph data structure and the terms dictionary. The dictionary is used to map vertex and edge labels (actually RDF terms) to unique integer identifiers. This lets us compress the graph representation in memory as well as perform label or term comparisons far more efficiently. Dictionary encoding is a common practice in database systems.

The SPARQL-to-C++ compiler is designed to operate on a shared-memory system and does not need any information about the physical partitioning of the database. However, as is common in PGAS libraries, GMT also exposes locality information, allowing for reducing 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 the runtime to spawn (or move) tasks on preselected nodes rather than

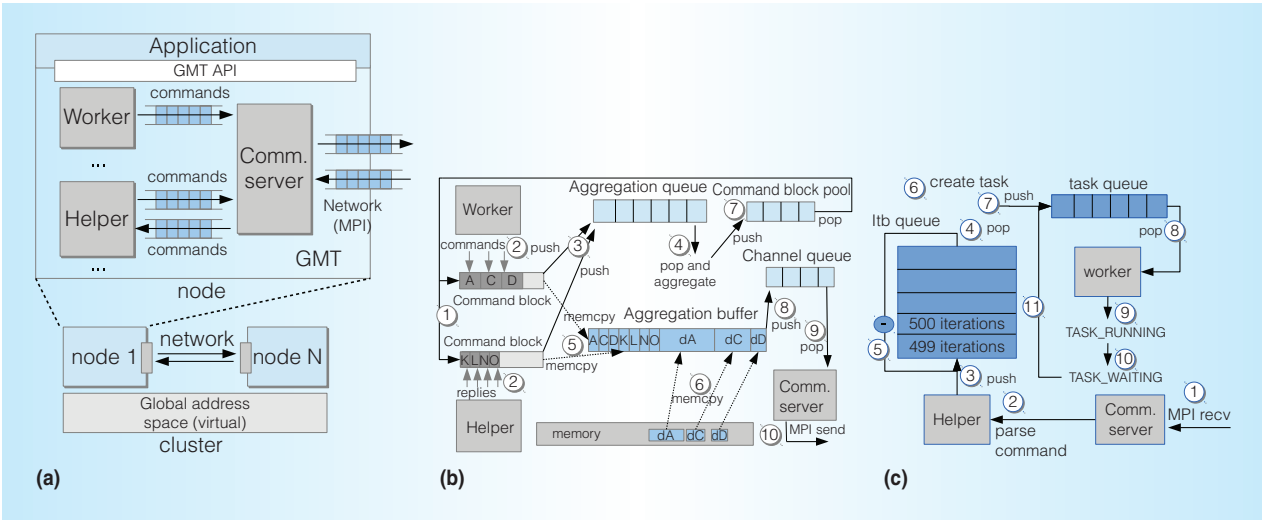


Figure 3. GMT runtime library. GMT architecture (a), aggregation (b), and fine-grained multithreading (c). The figure shows the general architecture of the runtime library with its specialized threads and the workflow for the two techniques (data aggregation and multithreading) that improve the behavior of commodity clusters with irregular workloads. (Comm. server: communication server; MPI: message passing interface.)

moving data. SGLib routines exploit these features to better manage SGLib's internal data structures. SGLib routines access data via *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 with both blocking and nonblocking semantics. GMT also provides atomic operations, such as atomic addition and test-and-set, on data allocated in the global address space. SGLib exploits these operations to protect parallel operations on the graph datasets and to implement global synchronization constructs for database management and querying.

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 must 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 toward the same destination nodes in aggregation queues. GMT copies commands and their related data (such as 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, nonblocking aggregation queues, which support concurrent access from multiple workers and helpers. Accessing these queues for every generated command would have a very high cost. Therefore, 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 preallocated command block from the command block pool (step 1). Command blocks are reused for performance reasons. Commands generated during program execution are collected into the local command block (step 2). A command block is pushed into aggregation queues when it is full (Condition A), or when it has been waiting longer than a predetermined time interval (Condition B). 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 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 (steps 4, 5, and 6). Aggregation buffers also are preallocated and recycled to save memory space and eliminate allocation overhead. After the copy, command blocks are returned to the command block pool (step 7). When the aggregation buffer is full, the worker (or helper) pushes it into a channel queue (step 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 (step 9) and performs a nonblocking MPI send (step 10). The aggregation buffer is then returned into the pool of free aggregation buffers.

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

Multithreading

Concurrency, through fine-grained software multithreading, allows GMT to tolerate 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 (that is, it gets a reply back from the remote node). In case of nonblocking commands, the task continues executing until it encounters a *wait* primitive.

GMT implements custom context-switching primitives that avoid some of the lengthy operations (for example, saving and restoring the 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 (step 1) 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 (step 2). The helper then creates an *iteration block* (*itb*). The itb is a data structure that contains the function to execute, the function's arguments, and the number of tasks that will execute the function. This way of representing a set of tasks avoids the cost of creating a large number of function arguments and sending them over the network. Next, the helper pushes the iteration block into the itb queue (step 3). Then, an idle worker pops an itb from the itb queue (step 5), decreases the counter of *t,* and pushes it back into the queue (step 6). The worker creates *t* tasks (step 6) and pushes them into its private task queue (step 7).

At this point, the idle worker can pop a task from its task queue (step 8). If the task is executable (that is, all its remote operations have been completed), the worker restores the task's context and executes it (step 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 (step 10) and is reinserted into the task queue for future execution (step 11).

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

Experimental results

We evaluated GEMS on the Olympus supercomputer at Pacific Northwest National Laboratory's Institutional Computing Center. Olympus is a cluster of 604 nodes interconnected through a QDR Infiniband switch with 648 ports (with a theoretical peak of 4 Gbytes/second [GBps]). Each of Olympus'

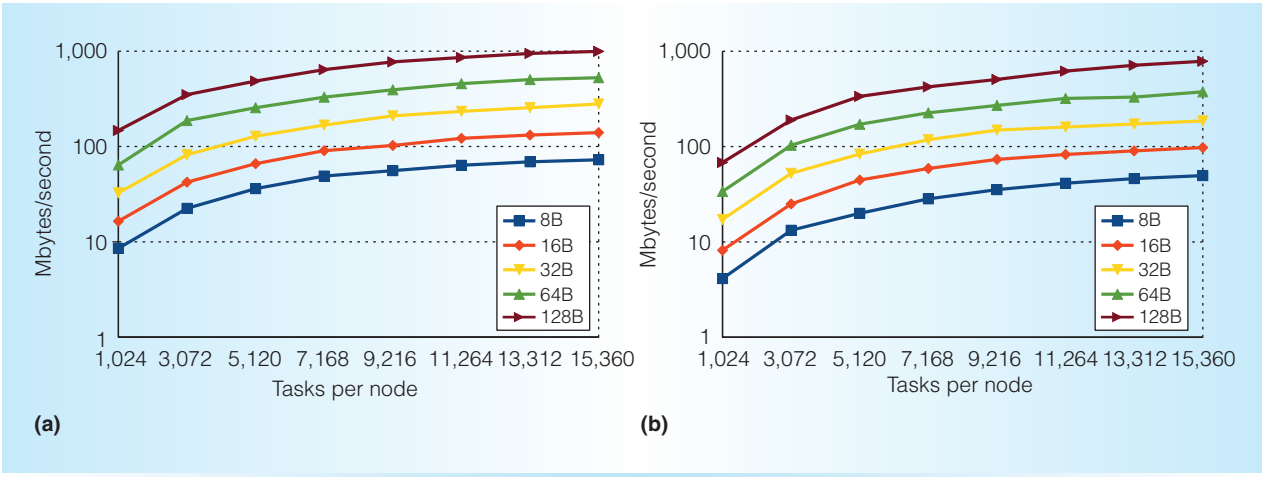


Figure 4. Synthetic benchmarks showing aggregation and multithreading effects. Transfer rates of *put* operations between two nodes (a) and among 128 nodes (one to all) (b) while concurrency is increased.

nodes features two AMD Opteron 6272 processors at 2.1 GHz and 64 Gbytes of double-data-rate 3 (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 one communication server per node. Each worker hosts up to 1,024 lightweight tasks. We measured the MPI bandwidth of Olympus with the Ohio State University Micro-Benchmarks 3.9 (http://mvapich.cse.ohio-state.edu/benchmarks), reaching a peak (around 2.8 GBps) with messages of at least 64 Kbytes. Therefore, we set the aggregation buffer size at 64 Kbytes. 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 system on a well-established benchmark, the Berlin SPARQL Benchmark (BSBM).[3]

Synthetic benchmarks

Figure 4 shows the transfer rates reached by GMT with small messages (from 8 to 128 bytes) when the number of tasks were increased. 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 are more messages that GMT can aggregate. For example, across two nodes (Figure 4a) with 1,024 tasks each, *puts* of 8 bytes reach a bandwidth of 8.55 Mbytes/second (MBps). With 15,360 tasks, GMT reaches 72.48 MBps. When we increase message sizes to 128 bytes, 15,360 tasks provide almost 1 GBps. For reference, 32 MPI processes with 128-byte messages reach only 72.26 MBps. 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 16-byte messages reach 139.78 MBps, whereas 32 MPI processes provide only up to 9.63 MBps.

GEMS results

The BSBM defines a set of SPARQL queries and datasets to evaluate the performance of semantic graph databases and systems that map the 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 1 through 6 of the Business Intelligence use case on datasets

Table 1. Time (in seconds) to build the database and execute Berlin SPARQL Benchmark (BSBM) queries 1 to 6 with 100 million triples. Execution times for queries are the average of 100 concurrent runs.

Phase/Query	2 nodes	4 nodes	8 nodes	16 nodes
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

Table 2. Time (in seconds) to build the database and execute BSBM queries 1 to 6 with 1 billion triples. Execution times for queries are the average of 100 concurrent runs.

Phase/Query	8 nodes	16 nodes	32 nodes	64 nodes
Build	628.87	350.47	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

with 100 million (100M), 1 billion (1B), and 10 billion (10B) triples.

Tables 1, 2, and 3 show the build time of the database and the average execution time (while running 100 queries concurrently) of the queries on 100M, 1B, and 10B triples, respectively, while we progressively increased the number of cluster nodes. Sizes of the input files are 21 Gbytes (100M), 206 Gbytes (1B), and 2 Tbytes (10B). In all cases, the build time scales with the number of nodes.

Considering all three tables together, we can appreciate how GEMS scales in dataset sizes when new nodes are added to the cluster, and how it can exploit the additional parallelism available. With 100M triples, Q1 and Q3 scale for all experiments up to 16 nodes. Increasing the number of nodes for the other queries, instead, provides constant or slightly worse execution times. These execution times are very short (under 0.5 seconds), and the small datasets do not provide sufficient data parallelism. These queries have only two graph walks with two-level nesting. Even with larger datasets, GEMS can already exploit all the available parallelism with a limited number of nodes. Furthermore, the database has the same overall size but is partitioned on more nodes, so 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, whereas the other queries, except Q1, approximately maintain stable performance. Q1 experiences the highest decrease in performance when 128 nodes are used because its tasks present higher communication intensity than the other queries, and GEMS has already exploited all the available parallelism with 64 nodes.

This data confirms that GEMS can maintain constant throughput when running sets

of mixed queries in parallel—that is, in typical database usage.

Our work on GEMS demonstrates that it is possible to implement a graph database on a commodity cluster that can scale in size while maintaining constant query throughput as new cluster nodes are added. GEMS shows that the typical issues that limit commodity clusters with graph processing (a typical irregular application) can be addressed using appropriate software techniques such as lightweight software multithreading and data aggregation. GEMS is a significant step toward the implementation of a fast, scalable, and cost-effective system for large scale graph databases.

We will continue to improve the stack by adding support to dynamic graphs and graphs that have multiple labels per edges (thick edges), by evolving the compiler to support the full SPARQL semantics and to perform more advanced query-planning optimizations. Furthermore, we will explore the support of lower-level communication libraries than MPI that directly map on the hardware (such as Infiniband verbs or the Cray custom primitives) and other emerging architectures for high-performance clusters in our runtime (such as accelerators and many-core processors), evaluating the power and performance tradeoffs with respect to commodity x86-based systems. MICRO

Table 3. Time (in seconds) to build the database and execute BSBM queries 1 to 6 with 10 billion triples. Execution times for queries are the average of 100 concurrent runs.

Phase/Query	64 nodes	128 nodes
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

Acknowledgments

This work was supported by the Center for Adaptive Super Computing Software (CASS) at the US Department of Energy's Pacific Northwest National Laboratory (PNNL). The Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830. A portion of the research was performed using PNNL Institutional Computing.

References

1. J. Weaver, "A Scalability Metric for Parallel Computations on Large, Growing Datasets (Like the Web)," *Proc. Joint Workshop Scalable and High-Performance Semantic Web Systems*, 2012, pp. 91-96.
2. J.R. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, 1976, pp. 31-42.
3. C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *Int'l J. Semantic Web and Information Systems*, vol. 5, no. 2, 2009, pp. 1-24.

Alessandro Morari is a research scientist in the Data Intensive Scientific Computing group at the Pacific Northwest National Laboratory. His research interests include big data analytics, large-scale runtime systems, system software for high-performance computing, and performance modeling. Morari has a PhD in computer science from Universitat Politècnica de Catalunya, Spain. He is a member of IEEE and the ACM.

Vito Giovanni Castellana is a research associate in the High Performance Computing Group at the Pacific Northwest National Laboratory. His research interests include embedded-system design and electronic design automation, code transformation, compilation, and optimization. Castellana has a PhD in computer science and engineering from Politecnico di Milano.

Oreste Villa is a senior research scientist in the architecture group at Nvidia Research. His research interests include computer architecture and simulation, accelerators

for scientific computing, and irregular applications. Villa has a PhD in computer science and engineering from Politecnico di Milano.

Antonino Tumeo is a senior research scientist in the High Performance Computing Group at the Pacific Northwest National Laboratory. His research interests include simulation and modeling of computer architectures (high performance and embedded), hardware-software codesign, FPGA prototyping, and GPGPU computing. Tumeo has a PhD in computer science and engineering from Politecnico di Milano. He is a member of IEEE and the ACM.

Jesse Weaver is a research scientist in the Data Intensive Scientific Computing group at the Pacific Northwest National Laboratory. His research interests include distributed graph and RDF databases, parallel reasoning systems, the Semantic Web, and linked data. Weaver has a PhD in computer science from Rensselaer Polytechnic Institute.

David Haglin is a senior research scientist in the Data Intensive Scientific Computing group at the Pacific Northwest National Laboratory. His research interests include data mining, big data, and graph algorithms. Haglin has a PhD in computer and information sciences from the University of Minnesota. He is a senior member of IEEE.

Sutanay Choudhury is a member of the Scientific Data Management group in the Computational Science and Mathematics Division at the Pacific Northwest National Laboratory. His research focuses on designing algorithms for modeling, mining, and searching streaming graphs to enable novel applications in social media, online news monitoring, and cyber security. Choudhury has a PhD in computer science from Washington State University. He is a member of IEEE and the ACM.

John Feo is the principal investigator of the High Performance Data Analytic Project and a deputy division director at the Pacific Northwest National Laboratory. His research interests include parallel computing, parallel application development, graph databases, functional languages, and performance studies. Feo has a PhD in computer science from the University of Texas at Austin. He is a member of the ACM.

Direct questions and comments about this article to Antonino Tumeo, Pacific Northwest National Laboratory, 902 Battelle Blvd MSIN J4-30, Richland, WA 99352; antonino.tumeo@pnnl.gov.