Porting the Urika-GD Graph Analytic Database to the XC30/40 Platform

Kristyn Maschhoff, Robert Vesse and James Maltby
Analytics R&D
Cray, Inc., Seattle, WA
{kristyn, rvesse, jmaltby}@cray.com

Abstract- **The Urika-GD appliance is based on a state of the art graph analytics database that provides high performance on complex SPARQL queries. This performance is due to a combination of custom multithreaded processors, a shared memory programming model, and a unique high performance interconnect.**

We will present our work on porting the database and graph algorithms to the XC30/40 platform. Coarray C++ was used to provide a PGAS environment with a global address space, a Cray-developed Soft Threading library was used to provide additional concurrency for remote memory accesses, and the Aries network provides RDMA and synchronization features. We describe the changes necessary to refactor these algorithms and data structures for the new platform.

Having this type of analytics database available on a general-purpose HPC platform enables new use cases, and several will be discussed. Finally we will compare the performance of the new XC30/40 implementation to the Urika-GD appliance.

Keywords- Urika-GD; Multithreading; PGAS; Semantics; SPARQL; RDF; XC30/40

I. INTRODUCTION

The Urika-GD appliance is a high performance semantic graph database based on the industry-standard RDF graph data format and the SPARQL graph query language. Urika-GD, launched in 2012, is based on the Cray XMT2 multithreaded shared-memory supercomputer. For the Urika-GD project we designed a high-performance graph database query engine that took advantage of the XMT2's unique hardware architecture. This query engine has been very successful, and offers the highest performance in the industry on complex queries over graph-structured data.

In order to expand the use of this form of analytics across Cray's product line, we have ported the Urika-GD query engine to the XC30/40 family of supercomputers. A new user interface has also been designed to take best advantage of the somewhat different operating environment. This new package, referred to internally as Cray Graph Engine (CGE), is nearing full functionality and we will present early performance benchmarks against the existing Urika-GD.

The remainder of the paper is organized as follows. The first section covers the Urika-GD appliance formulation and data format. The second section describes the process of porting the Urika query engine from XMT2 to XC in detail, including discussion of the PGAS programming model. The next section covers the relative performance data, followed by a section describing the user interface changes necessary for the XC environment. Finally, there is a description of use cases.

A. *What are SPARQL and RDF?*

RDF (Resource Description Format) [1] is a data representation first proposed by the World Wide Web Consortium (W3C) in 1999. It was originally intended to provide a flexible data representation for data across the World Wide Web, but it has proven to be very useful for description of any irregularly-connected data. The more common "relational" data model used in most SQL databases is based on tables, but RDF stores its data as a labeled, directed multi-graph. As a consequence, its data access patterns are very different than that of a standard SQL database. The basic unit of data storage in RDF describes a single graph edge and is referred to as a "quad," since it has four fields.

SPARQL is a query language designed to work with RDF databases, and it is also a W3C standard [2]. It is designed to look very similar to the well-known SQL query language, but with features designed to work with graphs as opposed to tables.

B. *Basic Graph Patterns*

The primary unit of search for the SPARQL query language is the Basic Graph Pattern, or BGP, instead of the row or column for tabular databases. A BGP consists of a connected set of nodes and edges that form a small graph. Names or types may be specified for some, all or none of the nodes and edges in the BGP. Finding all the instances of a BGP across a large graph is basically a subgraph isomorphism problem. A SPARQL query always starts with a BGP search, followed by additional filters, joins or other operators.

C. *Architectural issues with Graph analytics*

Because the data in an RDF database is graph structured, any node in the graph may share edges with any other set of nodes distributed across the memory of the system. Nodes may share edges with a few other nodes, or very highly connected nodes may have hundreds or thousands of immediate neighbors. This poses problems for distributed memory systems, since communication of information across the dataset is highly irregular, and may approach all-to-all for tightly connected graphs. Also, domain decomposition of graph data for load balancing is an unsolved problem in general.

D. *Shared Memory Implementation*

The ideal platform for a graph database of this type is a shared memory system with fast uniform access to all memory locations in the system. Luckily the Cray XMT / XMT2 provides this type of memory model. The Threadstorm processors on the XMT2 use a latency hiding technique [3] to create a large shared memory pool out of all of the compute node memory of the entire system, even though it is actually physically distributed. Thus, for the Urika-GD version of the database the graph is stored in a single largely one-dimensional data structure that spans the entire memory of the machine. Since memory access is effectively uniform, there is no need to partition or organize the graph for performance reasons.

E. *Multithreading on the XMT*

Because the search for BGPs is a highly parallel process, it is also desirable to have a high level of parallelism. The Threadstorm processors offer 128 hardware streams per node, and software threads may be allocated to the streams as needed. Since system sizes typically range from 64 to 512 nodes, thousands to tens of thousands of software threads may be employed for a given BGP search. This high level of parallelism is also useful for the other SPARQL functions.

II. PORTING THE URIKA-GD BACK END QUERY ENGINE TO XC

The Urika-GD back end query engine, which today runs on the Cray XMT2 architecture, performs several functions. First, the query engine is responsible for reading RDF, translating it into its memory-resident representation of the database, and writing the database to disk. Second, the query engine is able to read in a previously compiled database from disk in order to build its memory-resident database image. Third, it must accept SPARQL queries from the front end, evaluate them against the memory-resident database, and return the results to the front end. Fourth, it must accept SPARUL (SPARQL Update) commands from the front end and update the memory-resident database according to those commands. Finally, it must accept a checkpoint command from the front end and write the database to disk.

Although our initial porting focus was on being able to load in the database and run some basic SPARQL queries, each of the functions of the query engine mentioned above provided some unique challenges.

A. *Background on Porting Effort and Early Investigations*

As part of our early investigations into the feasibility of porting the Urika-GD appliance to the XC30, we studied two different approaches for porting the back-end query engine (the portion which today runs on the XMT2 architecture). The first was a “Soft-XMT approach” where we used an experimental memory domain library layer with soft-threads to emulate the shared memory programming environment of the XMT on the XC30. The second was an approach where we instead would rewrite the existing application to use a more standard HPC programming methodology, taking advantage of the global address space provided on the XC30 and the synchronization features of the Aries network. The advantage of the Soft-XMT approach would be minimal code changes from the existing Urika-GD query engine code base running on Cray XMT2 hardware today. As the pace of code development for Urika-GD was still very active, being able to either quickly pull over changes, or possibly maintain a single code base for both platforms, was highly desirable.

As a graph analytics application, maintaining optimal network performance for small word remote references, both puts and gets, was essential, thus we knew we wanted to leverage the low-level DMAPP library communication layer. Communication aspects of the query engine backend are similar to GUPS in many ways. The focus for the feasibility study was on the Basic Graph Patten (BGP) of the query engine. Initially we just extracted a time-dominant kernel of BGP in the Merge step described later and wrote both UPC and Coarray C++ implementations of this kernel. At the time we started our early investigation, the Coarray C++ project was also in early development as well. Although initially we were able to achieve better performance using UPC, working with Cray’s compiler team, we were able to eventually match UPC performance using Coarray C++. Converting the full Urika-GD application, which is predominantly C++, to use UPC, would have been an unwelcome challenge.

We were then able to compare performance for this simple kernel extraction to our “Soft-XMT” approach. One important observation we made at this point was that using a PGAS distribution of our internal data structures, we could very much benefit from locality, reducing the total number of remote references by as much as 75% for this simple kernel. The performance that could be gained by exposing locality led us to select Coarray C++ method over the “Soft-XMT” approach. This required more significant modifications to the Urika-GD source code, but in practice it turned out to be limited to a manageable number of sections. The Coarray C++ model provided us with the performance advantages of the low-level DMAPP communication layer (used indirectly via the PGAS library) and also the ability to take advantage of locality when available.

B. *Coarray C++*

The back end query engine of CGE is written as a distributed application using Coarray C++ as the underlying distribution mechanism. Coarray C++ is a C++ template library that runs on top of Cray's Partitioned Global Address Space (PGAS) library (libPGAS) and permits multiple processes (images) on multiple compute nodes to share data and synchronize operations. libPGAS is an internal Cray library supporting the Cray compiler, built on top of DMAPP. [4]

The sharing of data occurs using a symmetric heap, which is a virtual address space that is logically shared across all images. The logical sharing of the symmetric heap results in C++ symbols that refer to the same virtual address within the symmetric heap on all images. This implies that the order in which symmetric heap allocations are made is strictly ordered the same way across all images and that the size of allocations is strictly agreed upon across all images as well. In CGE we utilize both symmetric and asymmetric allocations for managing distributed data structures. Dynamically sized symmetric allocations, where all images allocate the same amount of memory from the symmetric heap in a collective manner, provide optimal performance. However, for memory saving considerations we also use coarrays of pointers, which then allow images to allocate memory independently (asymmetric allocations). In these situations the allocated memory is not likely to be located at the same address within every image's address space.

C. *Query Engine Execution model*

The coarray model of parallelism, at least at a high level, is a Single Program Multiple Data (SPMD) model. An advantage, from a program simplicity standpoint, of the SPMD model is that each image can be seen as executing serial code. This model turns out to be a good match to the current query engine execution model.

When the query engine receives a query, the query is first translated into a list of operators. We refer to this list as the "dispatcher list". It is a sequence of query engine operations, in Reverse Polish Notation order, to be performed by the query engine. Note that the sequential dispatcher list represents a design decision we made early on in XMT development not to parallelize any of the SPARQL operators at this level. For large-scale datasets, our assessment was that the overwhelmingly large source of parallelism would be data parallelism. Accordingly, each dispatched operator is implemented as a data parallel operation.

The dispatcher uses a simple stack to hold intermediate results between operations. We hold these intermediate results in Intermediate Results Arrays (IRAs). With the exception of Scan, which searches the database for matches to one or more quad patterns, all of the query engine operators input one or more IRAs, by popping them off the stack, and return one IRA, pushing it onto the stack. Every image holds an identical copy of the dispatcher list in local memory, and each image holds a distinct chunk of the IRA data locally.

The following class definition of the iResultArray class shows how we use coarrays to distribute the set of candidate solutions over images:

```
class iResultArray {
  public:
  int64_t global_numSolutions;
  coarray<int64_t> local_numSolutions;
  int64_t numUBV;
  int64_t* UBVlist;
  coarray<int64_t*> /* restrict */ solHuris;
…
}
```

Note that we use a coarray of pointers to store only the address of solHuris so that each image allocates only enough memory to hold the solutions it has been assigned.

Since we rely on data parallelism to maintain load balance, we periodically rebalance solutions within an IRA. The iResultArray class provides a rebalance operator which first rebalances solutions amongst images residing on the same physical node, and if still necessary, re-distributes solutions across all images.

The memory resident database consists of two primary global class objects, the dictionary and the DBase. The dictionary supports mapping between strings in the database to our internal integer representation. We refer to these integer representations as hashed URIs (HURIs). The Database class stores the RDF dataset in a more compressed and efficient format. The RDF data is represented as a set of graphs, where each graph contains the quads belonging to that graph

We use distributed hash tables for storing the dictionary and the Database. These will be described in more detail in a later section.

D. *Optimizing communication over the network*

The Basic Graph Pattern (also referred to as Quadpattern) consists of a sequence of three operations: Scan, Join, and Merge.

In the Scan step, we create an IRA for each query quad in the Quadpattern by searching through the local portion of the database residing on that image. Communication between images is only needed to balance the solutions across images.

To illustrate a common communication pattern found in multiple places in CGE, we will describe the Join operation in more detail. In the Join phase, we attempt to reduce the size of the IRAs returned by the Scan phase by comparing the variable bindings across multiple IRAs. For each unbound variable (UBV) that appears in multiple IRAs, we determine which HURIs are valid for that UBV. To be valid, a HURI must appear in at least one solution of all IRAs that contain that UBV.

To check for validity we use two integer arrays, masterA and tempA, which are globally distributed over all images. For simplicity assume these arrays are indexed by HURI. We initialize each element of masterA to 1 and each element of tempA to 0. We then update tempA by walking through the list of solutions, loading the HURI corresponding to a particular UBV (ubv_idx), and using this as an index to update the element of tempA. Here we use a simple block distribution for the IRA solutions so that we can easily compute which image and offset to write to.

```
 for (i = 0; i < this_ira->local_numSolutions; i++) {
   int64_t huri = this_ira->loadHuriElt(i, ubv_idx);
   int64_t image = huri / maxV_image;
   int64_t index = huri % maxV_image;
   tempA(image)[index]=1;
 }
 sync_all();
…
```

The call to the barrier sync_all() ensures that all puts have completed on all images. As written, this loop optimally issues non-blocking put operations using calls to libPGAS operations.

This is all managed in the Cray coarray_cpp.h template file. From a network performance standpoint, we use the desired non-blocking implicit (nbi) variants of the remote put operations, relying on the dmapp library optimizations. The above put operations use calls to the internal __pgas_put_nbi which directly uses the corresponding optimized dmapp_put_nbi operation.

A further optimization, used in multiple places throughout the CGE code base, is to first use local hashing to aggregate data to minimize network communication. The level of improvement obtained by first hashing locally is data dependent in terms of the relative size of the resulting number of local_keys compared to the local_numSolutions. The advantage is the smaller number of remote puts.

Following the sync_all(), we then iterate over tempA, and when this is not set to 1, we set the corresponding element of masterA to 0. Since we apply the same distribution to the masterA and tempA arrays, the load of tempA and the store to masterA are local operations on each image. Each image iterates over the HURI indexes assigned

```
IntHashTablel *local_table;
local_table = cqe_new(IntHashTable,
                this_ira->local_numSolutions);

for (i = 0; i < this_ira->local_numSolutions; i++) {
   int64_t huri =  this_ira->loadHuri(
                      i, ubv_idx);
   (void) local_table->insert(huri);
 }

 int64_t num_local_keys;
 int64_t* key_list;
  key_list = local_table->get_keys(&num_local_keys);

 for (i = 0; i < num_local_keys; i++) {
   int64_t huri = key_list[i];
   int64_t image = huri / maxV_image;
   int64_t index = huri % maxV_image;
   tempA(image)[index]=1;
 }
sync_all();
```

to that image.

Once this process is complete for each IRA, each element of masterA is set if the HURI is present in all IRAs or cleared otherwise. The last step is to iterate through each solution of each IRA. Any solution with a HURI that is not valid according to masterA is marked for deletion.

This requires a remote lookup of the elements of masterA for each of the HURIs in the local list of keys. In order to have these remote get operations performed using the desired non-blocking implicit get operations, we needed to explicitly call the get member function which uses __pgas_get_nbi and then use atomic_image_fence() which makes sure the get requests have completed. To fully benefit from using get_nbi operations, we also need concurrency. For this simple

```
key_list =  local_table->get_keys(&num_local_keys);
int64_t* key_flags = (int64_t*)
           cqe_malloc(num_local_keys * sizeof(int64_t));

 for (i = 0; i < num_local_keys; i += BLK) {
   int64_t blk_size = BLK;
   if(i+BLK > num_local_keys - 1)
         blk_size =  num_local_keys - i;

   for (int64_t k = 0; k < blk_size; k++) {
     int64_t huri = key_list[i+k];
     int64_t image = huri / maxV_image;
     int64_t index = huri % maxV_image;
     masterA(image)[index].get(&key_flags[i+k]);
    }
   atomic_image_fence();
 }

// Use key_flags to insert value into hash table for later
// lookup by huri

for (i = 0; i < num_local_keys; i++) {
  if(key_flags[i] == 1){
    local_table->set_data(key_list[i],key_flags[i]);
   }
 }
```

example, we simply use blocking so that each image can issue block_size get_nbi operations and then wait at a fence for these to complete. We then store the results from the remote lookup of masterA values into a local hash table for later use.

In this simple code example for Join, exposing the concurrency for remote gets could easily be achieved without using multi-threading. In fact, in simple situations like these, we just use a simple loop blocking technique as shown above. Software emulation of multi-threading becomes necessary when the individual get requests occur in deeply nested function calls.

E. Coupling Coarray C++ with a lightweight threading layer

The lightweight threading technology we use originally comes from the work of the Cray Chapel development team [5].

We have modified this for our own use such that each image starts up and maintains its own lightweight thread runtime. Each image independently generates concurrency by having each thread initiate a context switch following following any remote non-blocking get operation. At the node level the Cray dmapp library provides further optimization when scheduling these gets onto the network.

We have extended the Cray compiler-provided coarray_cpp.h template file to include additional member functions to allow us to issue a pgas_get_nbi call followed by either a thread context switch or a thread specific fence. Here instead of using the original get(), one would then use a get_switch().

For example, here are additional functions we added to the coarray_cpp.h file.

```
static void softthreads_switch_fence(void) {
  static int any_outstanding = 0; // global to all softthreads

  any_outstanding = 1;    // we just came from get/put_nbi()

  softthreads_switch();   // switch to next softthread

  if ( any_outstanding == 1 ) {
    // still something out on network;
    atomic_image_fence();
    any_outstanding = 0;   // anything out on network
                           // is now back
  }

  softthreads_switch();  // allow other threads to return and
                         // skip the atomic_image_fence()
}

void get_switch( pointer p, int64_t length ){
    internal::__pgas_get_nbi( p, m_image, m_addr, sizeof(
value_type )  );
    softthreads_switch_fence();
}
```

We needed a mechanism to be able to identify regions of code where we require multiple threads running in order to provide concurrency for deeply nested get() calls. This was accomplished by retargeting an existing OpenMP pragma recognized by the Cray compiler controlled via an environment variable.

Here is an example of how we have invoked threading in the function Order used to implement the SPARQL ORDER BY operation. We use threading here since this uses an get_switch() as part of fetching the solution HURIs.

```
#pragma omp parallel for schedule(static)
  for(int64_t i = 0; i < num_solns; i++) {
    int64_t tmp_offset =
              get_expr_offset(&sort_keys[(i*num_keys)]);
    int64_t image = tmp_offset / max_local_solns;
    int64_t image_offset = tmp_offset % max_local_solns;
    int64_t new_soln_offset = i * num_soln_vars;
    // Fetch the huris for this solution and store them
    // into the soln_huris.
    fetch_soln_huris(in_ira, image, image_offset,
                     num_soln_vars, sol_huris_ptrs,
                     &soln_huris[new_soln_offset]);
  }
```

FILTER and GROUP operations also make use of the threading layer in places where the individual get requests are buried deep in long call chains.

F. Extending/customizing Coarray C++ template files

For CGE we start with the coarray_cpp.h template file provided with the Cray compiler, but we have customized this to better suit our application requirements. In addition to adding threading specific member functions, we added othermember functions such as mget to simplify the syntax for multi-word gets. Another extension was to provide new member functions to allow vanilla loads and stores for coatomics so that we could make these loads and stores more lightweight in regions where we knew these values were not being updated.

G. Distributed Hash Tables

Probably the most utilized and important data structures in CGE are the various distributed hash tables. We use distributed string hash tables for storing the dictionary. We use distributed multi-word integer hash tables for storing the quads associated with each graph to allow enable fast insertions and deletions of quads. Distributed hash tables are also used to store expression results computed during the course of evaluating a query. Similar to the string dictionary, which maps strings to HURIs, we need to be able to efficiently map expression results to HURIs and HURIs to expression results.

One illustrative example is the application of the SPARQL operator DISTINCT which is used to remove duplicate solutions. The back-end query engine implementation of Distinct takes as input an IRA, removes duplicate entries while preserving the order of the remaining entries, and returns an IRA with the duplicates removed.

The basic strategy is to hash the solutions into a hash table, and note when we try to insert into the same location. A solution consists of a row of the IRA with the number of columns matching the number of variables. Since we need to preserve order, we insert both the key (multi-word key with the number of words being the number of variables in the IRA) and user data for each key. Here the user data keeps track of the original ordering.

The DistMwordHashTable class (Distributed Multi-word Hash Table) is an extension of our simplest DistHashTable (Distributed Hash Table) class. When an image inserts data (key, user data) into the global hash table, the image first inserts the data into a local table. Based on the key, a home image for that key is determined by the hash function. When the DistMwordHashTable sync() operator is called, the (key, user data) records are sorted into buckets based on the home image they belong to. Using coarrays, we communicate the number of keys each home node will receive from all of the other images, the pointers to the sorted keys and the sorted data, and the starting offsets. Each home image is then able to pull data from the other images and insert the keys into the global hash table. On the home image, we use double-buffering and non-blocking gets to overlap the insertion of one block of keys with the fetching of the next block of keys to insert. In the case where we have user data, if a key was not inserted (it was previously inserted), we update the user data for this key on the image sending the key with the data from the key already present in the table. This is done using non-blocking puts of the data from the home image back to the remote image.

H. How this is used in our implementation of DISTINCT

The first step is to allocate a multi-word global hash table using the DistMwordHashTable class with the total (global) number of solutions and the number of variables present.
One requirement for using the global hash table classes is that all images need to be involved when calling the constructors, destructors, and sync operations. Insertions prior to the global sync and lookup operations can be performed independently.

The global hash table includes a local host insertion table so that insertions are first hashed locally. This allows us to remove duplicates during the insert step before we sync with the other images, and before we send keys/data to the appropriate destinations.

```
for (int64_t soln=0; soln < numSols; soln++) {
   /* Insert mword key (huri) into global hash table*/
   int64_t* key = &theseHuris[soln*numUBV];
   int64_t soln_id = soln + soln_start;
   global_hash.insert(key,soln_id);
 }
 global_hash.sync();
```

The DistMwordHashTable class operator sync() is used to synchronize the local hash table with all the other images. This updates the global hash table, and then updates the user data, writing updated data back to into the local tables using remote Put operations. We use the updated user data to determine, in the case of duplicates, which of the solutions we keep based on which solution was inserted into the global hash table first.

Most of the interesting work and communication between images is handled within the sync() operation. Prior to pulling the local keys/data to their appropriate destination in the global hash table, we first sort the keys based on which image will receive them. We use symmetric dynamically sized coarrays to communicate the number of keys on each image and the starting location for each image to starting reading data.

```
coarray<int64_t []> image_key_counts(num_images());
coarray<int64_t []> image_key_starts(num_images());

coarray<int64_t *> sorted_keys(tmp_sorted_keys);
coarray<ght_user_data_t *> sorted_data(tmp_sorted_data);
```

We also save the pointers to the sorted keys and the sorted data in coarrays to allow the home image for these keys to access these since the number of keys to insert on each image may vary between images.

We have found that explicitly registering memory (registering the base address along with the full extent of the data that will be accessed remotely) avoids dmapp memory registration errors of the form:

```
ERROR: dmapp_mem_register(const_cast<void*>(addr),
length, desc ): The operation could not be completed due to
insufficient resources.
```

The registration currently uses an internal PGAS library registration function. It is helpful to register the base address and its extent with the PGAS library. This allows remote images to access offsets within that memory without registering the specific memory location each time. This helps prevent running out of resources for registering memory and can improve the performance of the remote accesses.

```
 __pgas_register(tmp_sorted_keys, (num_keys *
                     m_nwords * sizeof(int64_t)));
if(m_have_user_data) {
  __pgas_register(tmp_sorted_data, (num_keys *
                     sizeof(ght_user_data_t)));
}
```

We create a staggered ordering of images when getting the remote keys to insert into the home image hash table. We do this to avoid network contention so that all images do not all try to read data from the same image at the same time.

The DistMwordHashTable::get_next_keys member function fetch the data. This function issues non-blocking gets to allow overlap for fetching the next block of keys. It is up to the caller of this function to make sure the gets complete before using the fetched data.

One note on using this approach is that the allocation of the coptrs used to point to the remote addresses where we are to read from on the remote image is done in the caller, and these coptrs are passed into the function. This is necessary to make sure the coptr lives past the point where the non-blocking get completes (until atomic_image_fence() is called).

I. Summary of PGAS Porting Issues/Recommendations

CGE utilizes hugepages to better support the random small word communication patterns which are typical in graph analytics applications. We also turn off software ordering in the PGAS library in order to improve the performance of small-word puts and gets. This is done with some care, as the user is then responsible for making sure any ordering constraints are handled explicitly.

An additional parameter when running CGE on XC is managing the size of the symmetric heap. Although the launcher (cge-launch) sets this to a default value based on the amount of memory on the node and the number of images assigned to each node, depending on the size of the database and the memory requirements of the queries to be run, this may need to be adjusted explicitly as well.

The application dynamically allocates and frees blocks of memory of various sizes. With many images on each node competing for the same resources, this can be stressful for a parallel allocator. The default allocator, tcmalloc, was designed for performance. Unfortunately, the observed cost of this performance is that it can fragment memory in the face of many large (~32MiB +) allocation and free requests.

Two alternative allocators were tried: the system allocator from glibc and jemalloc. Neither of these allocators exhibits the fragmentation issues seen with tcmalloc, but they can be slower. Some queries took twice as long to evaluate. Other query evaluations were usually comparable to the tcmalloc implementation, but were occasionally significantly slower. Keeping in mind that the application tries to keep the data load-balanced among images, we chose to pre-allocate a fraction of available memory on each image. This memory is managed by a simple buddy allocator. Since each image has its own buddy allocator, the implementation is lock-free. Currently, the buddy allocator is used for larger allocations but tcmalloc is still used for smaller ones. While buddy allocators have a certain amount of internal fragmentation, this design is very effective at returning memory to its fully coalesced state after each query. At the same time, its performance is comparable to tcmalloc.

Table 1 PGAS Porting Recommendations

Issue	Recommendation
Improve performance of remote PUTs and GETs	Disable SW Ordering by setting the environment variable setenv PGAS_NO_SW_ORDERING 1 Any ordering constraints must be handled explicitly by the user.
Increase concurrency for remote GETs	Issue multiple non-blocking operations before issuing an image_fence
Expose concurrency for remote GETs in deeply nested calls	Using lightweight threading layer to increase the number of non-blocking GETs issued.
Reduce latency due to waiting on GET requests to complete	Overlap computation and communication
Asymmetric coarrays incur additional overhead for remote address lookup	When possible, use symmetric coarray allocations.
Improve performance of remote gets	Cache coptrs to remote memory
Avoid dmapp resource errors	Explicitly register memory with PGAS
Memory fragmentation using tcmalloc	Preallocate memory to be managed by the application, using buddy allocator

III. Performance of CGE vs. Urika-GD

Despite the changes in memory model, we have experienced generally improved performance with CGE on XC30/40 vs. Urika-GD. The benchmark data presented here is for equal number of nodes of Urika-GD and on XC40. The test cases are the two most widely used SPARQL benchmarks, Lehigh University BenchMark (LUBM) [6] and SParql Performance Benchmark (SP2B) [7]. Performance on the LUBM benchmark concentrates primarily on the Basic Graph Pattern (BGP), which is very communications intensive and tests the efficiency of the parallelization. Figure 1 and Table 2 show execution times for LUBM25K on 64 nodes. LUBM25K is a moderate-sized benchmark, around 3 billion quads. Performance for the XC40 was generally superior to Urika-GD, particularly on the most complex query, query 9.

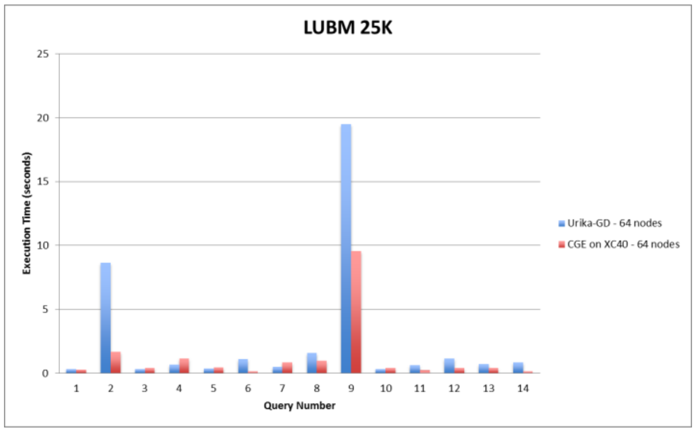

Figure 1. LUBM25k results on 64 nodes

Query number	Urika-GD	CGE-XC	Speedup
1	0.31	0.28	1.12
2	8.66	1.66	5.22
3	0.31	0.39	0.81
4	0.68	1.14	0.59
5	0.38	0.43	0.86
6	1.11	0.13	8.79
7	0.51	0.86	0.59
8	1.61	0.97	1.66
9	19.48	9.57	2.04
10	0.31	0.40	0.77
11	0.61	0.24	2.53
12	1.15	0.43	2.69
13	0.71	0.41	1.72
14	0.85	0.13	6.67

Table 2. LUBM25k comparison, 64 nodes

Performance for a 4x larger benchmark, LUBM100k, is shown for 256 nodes in Figure 2 and Table 3. Performance is roughly equivalent on query 9, but still better on simpler queries. Additional optimizations to improve scaling are in development.

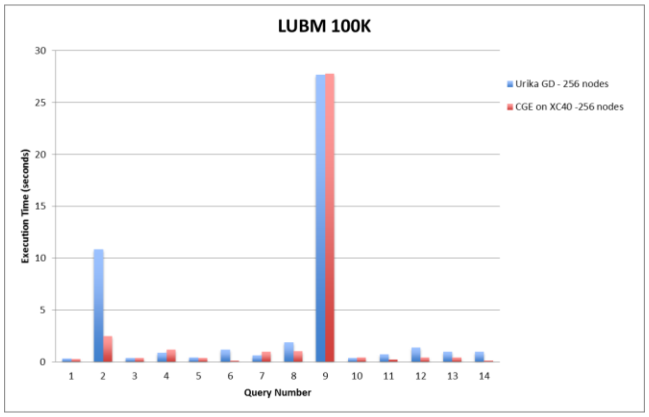

Figure 2. LUBM100k results on 256 nodes

Query number	Urika-GD	CGE-XC	Speedup
1	0.35	0.28	1.25
2	10.87	2.51	4.33
3	0.36	0.36	1.00
4	0.87	1.17	0.74
5	0.42	0.40	1.05
6	1.19	0.13	9.54
7	0.64	0.98	0.65
8	1.89	1.05	1.80
9	27.68	27.79	1.00
10	0.36	0.44	0.81
11	0.72	0.24	2.95
12	1.38	0.44	3.12
13	1.00	0.42	2.39
14	0.98	0.13	7.63

Table 3. LUBM100k comparison, 256 nodes

Performance on the SP2B benchmark is more dependent on the other SPARQL operators such as FILTER, DISTINCT and ORDER BY. As such, it is less sensitive to communications and more sensitive to node compute power. Figure 3 and Table 4 show results for SP2B500m on 256 nodes.

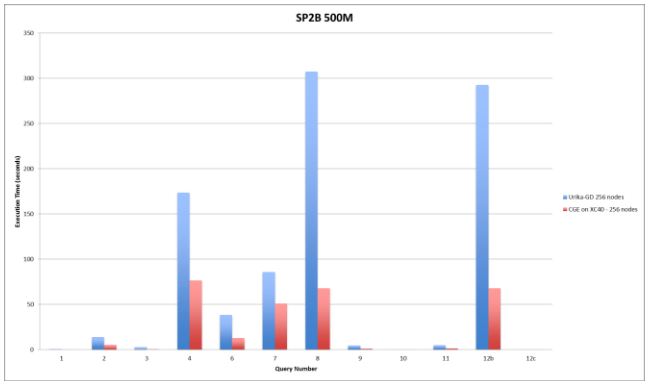

Figure 3. SP2B500m results on 256 nodes

Query number	Urika-GD	CGE-XC	Speedup
1	0.50	0.14	3.66
2	14.05	5.31	2.65
3	2.67	0.43	6.25
4	173.80	76.57	2.27
6	38.18	12.73	3.00
7	85.75	50.93	1.68
8	307.62	67.93	4.53
9	4.41	0.79	5.59
10	0.23	0.01	15.17
11	4.98	1.39	3.57
12b	292.89	67.87	4.32
12c	0.21	0.03	8.16

Table 4. SP2B comparison, 256 nodes

Performance is considerably better on the more powerful nodes of the XC40.

IV. User Interface Porting Issues

In porting the database server from Urika-GD, which is a hardware appliance, to the XC30/40 platform we had to rethink our user interface paradigm. The existing Urika-GD product is designed as an appliance so we provide a central management interface known as the Graph Analytics Manager (GAM) on top of the database server. GAM provides users with access to various functionalities for both analytics and system administration depending on the users' permissions. Much of this functionality was designed around the appliance nature of the system i.e. it aimed to provide a single pane of glass for interacting with Urika-GD. As a result we knew there would only ever be one GAM instance running and we were therefore able to make a lot of assumptions about file system layout, network ports for communications, and how the user would interact with the system.

In porting to XC30/40 it was clear that this paradigm was no longer appropriate for what was intended to become a user application. Thus we needed to allow for potentially many instances of the software to be running on the system such that they wouldn't interfere with each other. This implied a number of design constraints for the port:

- All components needed to be able to communicate on arbitrary ports that could be user specified
- User interface needed to be much more lightweight and minimal than the Urika-GD interface
- User interface needed to permit batch style interactions i.e. command line and potentially non-interactive

With these constraints in mind we designed a new user interface that is command line based with an optional minimalist web interface launched via the same command line. This web interface is primarily needed to provide the ability for the port to be accessed via standard compliant SPARQL endpoints by any SPARQL capable tooling.

A. *User Interface Components*

The user interface can be divided into two main functionalities: job launch and user interaction. These are provided by the cge-launch and cge-cli commands respectively.

The cge-launch command is responsible for launching the CGE server and providing a well-known port to which the user interface can be connected. The launcher is capable of interacting with various different job schedulers, currently SLURM and Torque/MOAB, with the ability to support other job schedulers in the future if necessary. This allows the CGE server to be launched across XC30/40 systems configured with different job schedulers using a consistent command line interface. Once the CGE server is launched the launcher detects the compute node host and port on which the server is listening and sets up forwarding from the user's desired (and possibly externally visible) port to the internal host and port.

After the CGE server is launched the cge-cli command (the Command Line Interface (CLI)) can be used to interact with it as desired. This command provides multiple sub-commands that can be used to perform different actions on the database including both analytic interactions (queries and updates) and administrative actions (configuration and shutdown). Additionally as already noted it can be used to launch a web interface that provides standards compliant SPARQL endpoints [8] such that SPARQL aware tools can communicate with the database.

The cge-cli command is designed to allow operation in both interactive and non-interactive modes. In non-interactive mode some functionality is more limited but it allows interactions with the CGE server to be incorporated into batch scripts as desired.

B. *Network Communications*

One limitation of the XMT platform is that it isn't possible to do ad-hoc TCP/IP connections between the login and compute nodes due to the unique hardware and software kernel present on the compute notes. Instead we use the Cray Lightweight User Communication (LUC) [9] library to create a persistent long-lived connection between GAM and the database server across which RPC calls could be made. In moving to the XC30/40 platform we were able to take advantage of a standard Linux OS with a standard TCP/IP and sockets stack that allowed us to rewrite portions of the communications between the UI and the database engine.

Since portions of our existing RPC calls from Urika-GD already ship complex serialized data structures across the network for the XC30/40 platform we chose to wrap these existing data structures in a simple binary protocol. This protocol reads and writes messages to and from the network using a compact header to instruct the receiver of the nature of the message. Each message type is capable of carrying an appropriate payload that may be a binary data structure, command arguments or empty.

The main complexity that this introduced in the port was the need to have a listening socket in a PGAS application where for the most part the images work in lock step. We only needed one listening socket so it was necessary to architect the application such that only a single image would be responsible for managing the socket. We chose to use Image 0 as our leader. We set up the socket there on application start up and then this image waits to receive messages from users. The other images are blocked at a sync_all() waiting for Image 0 to reach it while Image 0 is waiting for messages. On receipt of a message it validates the message and then proceeds to the sync_all() call; this wakes the other images and they then all proceed to process the message in parallel. Finally Image 0 again becomes the sole process left running as it sends the appropriate response back to the user before proceeding to wait for the next message.

C. *Security*

Moving from an appliance to a user application also implied that users would manage the security of their own databases instead of relying on appliance administrators. This meant that we needed to consider how users would interact securely with their databases and how they would be able to grant access to their databases to other users.

At the most basic level database security is managed via standard file system permissions. A database lives in two locations on the system: the data directory and the results directory. The data directory contains the database's actual stored data in binary form while the results directory contains the results of queries made against the database. If Alice locks down the permissions on her data directory then Bob is not able to access that database. This means that he cannot take a copy of the database nor can he start up a CGE server against that database.

The results directory is usually a different directory and Alice may wish to allocate this directory or specific results files within it more open permissions in order to share results with other users. However users must be careful in doing this because sharing some query results may be sufficient for another user to recreate his own version of the database.

D. *Securing Network Communications*

It should be clear to any sufficiently deviously minded reader that our initial approach to network communications had some security holes in it. At first it was entirely possible for Bob to run commands against a database launched by Alice since we imposed no form of access controls. This meant that Bob could unintentionally or maliciously interfere with Alice's database e.g. adding/deleting data or accessing supposedly secure data. Also since we were using plain socket communications any sufficiently privileged user could eavesdrop upon or perform man in the middle attacks on communications between any user and CGE server on the system.

To address these issues we decided to wrap our communications protocol inside of an encryption based authenticated security protocol. Of the various choices available the SSH protocol seemed like the one that would be most familiar to users as most would already have had to set up SSH access to Cray systems. There were also library implementations of the SSH protocol available for both the server and client implementations making for easy integration. Thus we implemented an SSH protocol transport as a secure wrapper around our own communications.

Using this secure transport between CLI and server gives us several advantages:

- The owner of a database can completely control what other users have access to his or her data via the CLI commands.
- Connections from the CLI to the server simply fail to establish if the invoking user does not have an authorized identity key,
- Connections are established without the use of passwords, making non-interactive use of the CLI possible,
- Users are protected from eavesdropping, man-in-the-middle, and other kinds of attacks at least on CLI to Server interactions.
- Users are authenticated and identified by the protocol, so decisions can be made by the server based on who the requesting user is.

To keep setup of our SSH transport simple from a user's point of view, we decided to use the existing ~/.ssh directory files already present for most users for basic authentication. If Alice only wants to be able to interact with her own database and is not interested in granting other users access, she need only make sure that her SSH keys are set up to allow her to use the ssh command to log into localhost. With this configuration, Alice can be certain that no other user can connect a CLI command to her server, and she can connect CLI commands at will.

Going beyond this, if Alice decides to allow Bob access, she has a choice:

- Grant access to all of my databases,
- Grant access to only some of my databases.

To grant Bob access to all databases, Alice can add Bob's public identity key to an authorized_keys file in her ~/.cge directory. Bob will now be able to connect a CLI with any server launched by Alice.

To grant Bob access to only one database, Alice can add Bob's public identity key to an authorized_keys file in the data directory containing the database she wants Bob to be able to use, and take Bob's public identity key out of the authorized_keys file in her ~/.cge directory. Now Bob will only be allowed to connect to a server that is serving the database in that particular data directory. Of course, Alice can add Bob's key to as many data directory authorized_keys files as she likes.

Part of the SSH protocol is a mechanism allowing the server to authenticate itself to the client. This requires the host to present a "Host Key" to the client, which the client verifies matches what is expected. This allows a client to decline to talk to an untrusted server.

In the simplest case, we solve this problem by using the SSH public identity key (taken from the user's ~/.ssh directory) of the user that launches CGE as the CGE server Host Key. This should be sufficient in almost all cases, but there may be situations in which a user needs to use a CGE specific host key. In that case, the user can use ssh-keygen to create a key-pair in his or her ~/.cge directory and that will be used instead.

Recognition of the Host Key by a client is based on entries in the client's known_hosts file. Conveniently, Host Keys for hosts on non-standard SSH ports are differentiated from other Host Keys from the same IP address by the TCP/IP port number used by the server. This means that, as long as a user does not use the standard SSH port to serve access to a database, the normal SSH host key and the CGE host key can be different and coexist.

The trusted server aspect of the SSH protocol is there to protect secrets that might be exchanged between the client and server against man-in-the-middle snooping, where a snooper is interposed as follows:

CLI <---> snooper <---> Server

In this kind of attack, the snooper looks like an SSH server (but has the wrong host key) to the CLI and looks like an SSH client (using some other authorized public identity key) to the server. The snooper is then free to look at clear-text data, most notably user passwords, before passing traffic between the CLI and Server.

Because of this SSH clients generally verify the host key and complain interactively when they don't recognize the server. This means that the first connection with any new server must be made interactively, and, if the key changes subsequent connection attempts will fail. For CGE, this potentially poses a challenge, in that Alice might choose to start her server on port 12345 today and on port 23456 tomorrow. This would require interactive re-validation of Alice's Host Key each time, which could become cumbersome.

Fortunately, in the case of CGE, the security concern about man-in-the-middle attacks is greatly attenuated by two facts:

- We do not allow the use of passwords for authentication, and
- Any snooper must have been started by an already authorized user, who would already have any access that the CLI user would have, so there is no chance of unauthorized disclosure of database content through the snooper.

This means that, while we have to use Host Keys to complete the SSH protocol, there is minimal need for server trust in our implementation. Therefore the CLI may optionally be configured to automatically trust new host keys presented to it.

The final advantage of using an authenticated protocol is that the identity of the user invoking the CLI command is known to the server in a way that cannot be spoofed. This means that the server can use that identity to make security related decisions on behalf of the owner of the database. While our use of this is limited at present since we currently restrict shutting down a CGE server to the user who started the server, it opens up the possibility of more detailed security policies, all under control of the owner of a database, in the future.

E. Leveraging Parallel IO

On the Urika-GD platform the systems are attached to a fast parallel file system (usually Lustre), which is mounted on the system. Parallel IO for input and output by the database server is achieved using the Cray libsnapshot library for XMT. This handles dividing up the IO tasks across all XMT compute nodes and performing the IO in parallel through the appropriate service nodes.

On the XC platform, access to fast parallel file systems is globally available on compute nodes through standard POSIX file operations. This means we do not need the libsnapshot mechanism for IO. Instead, when file IO is required, individual images on individual nodes can parcel out a large IO operation amongst themselves and do the IO directly in parallel.

While this greatly simplifies parallel IO, it raises a concern about saturating available throughput between a compute node and the file system. Since we run multiple images per node in the CGE server, and a single image has sufficient power to saturate the file system IO available to that node, if we allowed all of the images on the node to do parallel IO, the IO congestion would likely degrade IO throughput.

To avoid congestion, we created the concept of an Image Group, in which all images residing on a given node are in the same group, and there is a group leader image, defined as the image with the lowest numbered image ID in the group. Using this construct, we are able to limit the IO on a given node to an image (the image group leader) and take advantage of locality to carry out all of the IO that would otherwise have been assigned to the other images in the group. By taking this approach to parallel IO, we have been able to make use of available IO bandwidth.

V. Use Cases on XC30 / 40

The benefit of having this advanced graph analytic capability as part of a large, multi-purpose computer system like the XC30/40 is the ability to enable complex, multi-step workflows on a single computer system, reducing data movement and decreasing time to solution.

A good example of a multi-step workflow is the Institute for Systems Biology (ISB) drug repurposing analysis. This analysis consisted of three highly compute-intensive steps. First, a Natural Language Processing (NLP) analysis was performed on the Medline literature database, using a thousand-node cluster. Next, a Random Forest Classifier was applied to experimental genomic data from The Cancer Genome Atlas (TCGA), to quantify experimental relationships between genes in cancer. This analysis was performed on 200,000 nodes of Google Compute. The data from those two analyses was converted into RDF format and combined with other public RDF databases about known drugs and pathways and loaded into Urika-GD. Urika-GD was used to discover new relationships amongst drugs, genes and pathways, and identify candidates for drug repurposing. It was successfully used to identify an HIV drug that may actually be effective against breast cancer.

This analysis was performed on three different high performance computer systems, but with the advent of CGE they could all be performed on a single XC30/40 system.

VI. Conclusions

The Urika-GD semantic Graph database has successfully been ported to a new architecture, that of the XC30/40. This was made possible by taking advantage of the Coarray C++ programming model available on the XC and its Aries network. Performance on the XC30/40 is significantly improved over the XMT2-based Urika-GD, with the performance improvement varying with the nature of the benchmark query. Further scaling work is planned, and improvement is certainly possible.

Moving the semantic database from a dedicated appliance to a shared, batch scheduled HPC environment like the XC30/40 required a significant redesign of the user interface, as well as reworking of the security model. However, we believe that the Cray Graph Engine will prove to be extremely useful for large, mixed analytics workflows on HPC systems.

ACKNOWLEDGEMENT

This work described would not be possible without the contributions of many individuals on the CGE project: Jim Harrell, Andy Kopser, Eric Lund, David Mizell, Chris Rickett, Mike Ringenburg, Tom Webber and Karlon West. We also would like to thank Troy Johnson and Steve Vormwald for supporting our porting efforts, and providing us with updated PGAS libraries as needed.

REFERENCES

[1] "Resource Description Framework (RDF): Concepts and Abstract Syntax," http://www.w3.org/TR/rdf-concepts/. W3C Rec. 03/2014.

[2] "SPARQL Query Language for RDF," http://www.w3.org/TR/sparql11-query/. W3C Rec. 03/2013.

[3] Andrew Kopser and Dennis Vollrath, "Overview of the Next Generation Cray XMT,", Cray Inc., Cray User Group 2011 Proceedings

[4] Johnson, T. A., Coarray C++. In 7th International Conference on PGAS Programming Models, Edinburgh, Scotland (2013)

[5] Brad Chamberlain, Sung-Eun Choi, Martha Dumler, Tom Hildebrandt, David Iten, Vass Litvinov, Greg Titus, Casey BaAaglino, Rachel Sobel, Brandon Holt, Jeff Keasler "Chapel HPC Challenge Entry, " SC12, (2012)

[6] Guo, Yuanbo, Pan, Zhengxiang and Heflin, Jeff . LUBM: A Benchmark for OWL Knowledge Base Systems. Web Semantics. 3(2) July 2005. pp.158-182.

[7] Michael Schmidt, Thomas Hornung, Georg Lausen and Christoph Pinkel, "SP2Bench: A SPARQL Performance Benchmark" Proc. ICDE'09 (2009)

[8] Lee Feigenbaum, Gregory Todd Williams, Kendal Grant Clark, Elias Torres eds. – 2013 – W3C - http://www.w3.org/TR/sparql11protocol/ Retrieved 12th March 2015

[9] Cray XMT Programming Environment User's Guide, Sept. 2012, S-2479-202