Optimizing the Cray Graph Engine for Performant Analytics on Cluster, SuperDome Flex, Shasta Systems and Cloud Deployment

Christopher D. Rickett, Kristyn J. Maschhoff, Sreenivas R. Sukumar
Hewlett Packard Enterprise

chris.rickett@hpe.com
kristyn.maschhoff@hpe.com
sreenivas.sukumar@hpe.com

Abstract—We present updates to the Cray Graph Engine, a high performance in-memory semantic graph database, which enable performant execution across multiple architectures as well as deployment in a container to support cloud and as-a-service graph analytics. This paper discusses the changes required to port and optimize CGE to target multiple architectures, including Cray Shasta systems, large shared-memory machines such as SuperDome Flex (SDF), and cluster environments such as Apollo systems. The porting effort focused primarily on removing dependences on XPMEM and Cray PGAS and replacing these with a simplified PGAS library based upon POSIX shared memory and one-side MPI, while preserving the existing Coarray-C++ CGE code base. We also discuss the containerization of CGE using Singularity and the techniques required to enable container performance matching native execution. We present early benchmarking results for running CGE on the SDF, Infiniband clusters and Slingshot interconnect-based Shasta systems.

Index Terms—graph analytics, semantics, PGAS, parallel programming, pattern search, pattern mining, Cray Graph Engine.

1 Introduction

Since the HPE acquisition of Cray, the focus of the Cray Graph Engine (CGE) has expanded from a solution limited to the Cray XC hardware platform, to providing a performant solution across multiple architectures and cloud environments (on-premise or as-a-Service) in order to support the growing market adoption of graph databases and tools in the public cloud. The focus of this paper is on the modifications made to the Cray Graph Engine to enable both portability and containerization, while maintaining performance.

The importance and influence of graph analytics in high-performance workloads continues to grow and there is increasing customer demand for scalable solutions for analyzing graphs. We continue to see demand in the areas of fraud detection; cybersecurity, but more recently have seen growing interest in the areas of knowledge graph traversal, recommender systems, etc. In response, there has also been a flood of options available in the market such as Neo4j, TigerGraph, AnzoGraph, BlazeGraph, GraphX, Apache Giraph, Trovares etc. While most of these databases have seen enterprise wide adoption due to their accessibility via public cloud instances, customers are still looking for better solutions that (i) are cost-effective at data sizes greater than 2 TB, (ii) execute queries in the order of seconds for interactive vertex-centric basic-graph pattern searches and graph-theoretic whole-graph analysis algorithms, (iii) is a graph database that horizontally scales-out and vertically scales-up, and (iv) can be run in a container on their laptops for application development and then transitioned into production on specialized clusters and supercomputers as data sizes grow.

The key to efficient performance for CGE, and the focus of several Cray User Group papers discussing optimizing CGE performance on Cray XC systems [1, 2, 3], is how one best utilizes the remote direct memory access (RDMA) capabilities in the network to maximize efficient remote access of data. In this paper we demonstrate how efficient RDMA communication is also achievable when running CGE from within a container and we discuss the techniques required to enable container performance to match that of native execution. While the focus of this paper is on containerizing CGE to enable optimal performance on multiple platforms, the techniques discussed are also applicable to containerizing other RDMA dependent high-performance computing (HPC) applications.

With this latest set of updates, CGE can now be deployed across a variety of architectures including the HPC Cray Shasta systems, large shared-memory machines such as HPE SuperDome Flex (SDF), and cluster environments such as HPE Apollo systems.

2 Background

The first semantic graph database developed at Cray was the Urika-GD database appliance, launched in 2012. The scalable shared memory model of the Cray XMT2 and its

high performance indirect addressing enabled connections in the graph to be searched efficiently, and the multi-threaded programming model made it possible to access data with high-concurrency to hide latency. The Urika-GD database was ported to the XC30 architecture in 2015, as described in a previous CUG paper [1]. This was accomplished by making use of the Partitioned Global Address Space (PGAS) programming model and Coarray-C++. The back end query engine of CGE was refactored 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 and permits multiple processes (images) on multiple compute nodes to share data and synchronize operations. The Cray PGAS library supports the Cray compiler and is built on top of DMAPP [4] which targeted the Aries interconnect. Further optimizations to CGE were described in a 2017 CUG paper [2] and also in a 2018 CUG paper [3] which described the additional optimizations made to CGE to achieve scalable high performance on a very large database consisting of over one trillion Resource Description Framework (RDF) [5] triples.

3 Our Solution

As previously discussed, CGE was written using Cray's Coarray-C++ which is based upon the Cray PGAS library that leverages the Cray DMAPP library targeting the Aries interconnect. In order to port CGE to enable efficient execution on multiple platforms, the first step was to remove the dependencies on the Cray hardware and software stack. The design of CGE encapsulates nearly all of the details about the communication layer for remote data access and image synchronization within the Coarray-C++ and PGAS implementation. This meant that if the Cray Coarray-C++ and PGAS software stack could be replaced with a platform independent solution then CGE would be able to execute across multiple platforms with limited changes to CGE itself.

The first and largest part of replacing the Cray software stack was in creating or finding a replacement PGAS library that could run on multiple platforms. The first step done was to look at the features required of CGE for the PGAS library since not all of the existing features are used. The primary features of the PGAS library that CGE required are:

- creation of a symmetric virtual address space across images
- symmetric heap malloc/free functions
- blocking/non-blocking one-sided puts/gets
- barriers
- collectives (i.e., sum/min/max)
- direct memory access amongst images on same physical node

In CGE, images that reside on the same physical node are referred to as a group. A key design feature of CGE is the group-aggregation of messages, where data is aggregated within images in a group first so that all data for/from a given target image can be put/get with a single message whenever possible. This has been shown to significantly improve the scalability of CGE by increasing the overall size of messages but drastically reducing the number of messages. For common operations such as scan/join/merge, this has been shown to make the operators ∼3x faster than when all images performed more all-to-all style communications [3]. In order to enable this group aggregation, the PGAS replacement would need to provide a mechanism for an image to directly read/write memory for other images on the same physical node.

Another key detail about CGE that impacts the interaction with the PGAS library is that CGE manages its own memory, including the symmetric heap allocations. This was done early in the creation of CGE to address memory fragmentation problems that arose when using the system malloc, which is based upon *tcmalloc* [6], that lead to CGE hitting out of memory errors when running queries repeatedly. This was due to a combination in how CGE uses memory and how *tcmalloc* caches memory for faster allocations of similar sizes later [6]. This caching can cause problems when trying to coalesce memory and resulted in CGE being unable to allocate large blocks of memory after running a number of queries. To address this limitation, CGE uses its own memory allocators to handle allocations of different types, such as temporary, persistent or symmetric. Enabling CGE to manage its own symmetric memory adds an additional requirement for the PGAS library.

3.1 Simplified PGAS

Given the desire to port CGE to execute efficiently on multiple platforms, including in a cloud environment using containers, our solution was to replace the Cray PGAS with a simplified one of our own that implements only the necessary features. To accomplish this, we based our PGAS library on POSIX shared memory [7] and the Message Passing Interface (MPI) [8]. The primary motivation of using these two standards is to increase the number of platforms that CGE could execute efficiently on while reducing the amount of effort required to create the simplified PGAS library. Performant MPI implementations are available for a large number of architectures and can easily be containerized, greatly expanding the potential architectures where CGE could perform well.

3.1.1 Symmetric Heap

The symmetric heap is created on each image using POSIX shared memory with all images mapping their heap at a known starting address to easily enable calculations of addresses for remote gets/puts. An instance of the CGE memory allocator is created as part of the PGAS library that maps the POSIX shared memory and this allocator handles any symmetric allocation/free requests by an image. Reusing the existing CGE memory allocator class to serve the symmetric heap allocations further simplified the effort required to develop the library. Previously, the size of the symmetric heap in CGE was only a fraction of the available memory on a node but with the new PGAS the symmetric heap is used to create all of the memory that an image will use. The symmetric heap is then used to provide memory for the other allocators used by CGE. As will be discussed below, this enables memory to easily be shared amongst images in a group but has a drawback in that the default size

of */dev/shm* used for POSIX shared memory is only 50% of RAM, which can significantly limit the amount of data CGE can store per node. However, this default size can easily be increased with a remount of */dev/shm* to a larger percentage of RAM. In our tests we usually remount it to be 85% of RAM but this could be tuned based on user requirements.

The Cray Coarray-C++ and PGAS provided a function, *to_local()*, that could be used to return a local pointer to an image for memory owned by another image on the same physical node to enable direct reads/writes. This required XPMEM to enable the mapping of virtual memory of one process into the virtual address space of another on the same node [3]. XPMEM requires a kernel mod as well as a user space library, so leveraging XPMEM in the simplified PGAS would limit portability and containerization. The simplified PGAS uses POSIX shared memory to enable mapping of the virtual address spaces to enable direct reads/writes of memory between images on the same node. The shared memory segments of images on the same physical node are mapped into each other's virtual address spaces at program startup and the new PGAS library provides the same *to_local()* function for creating a local pointer to memory owned by another image within an image's group. Using POSIX shared memory to enable direct memory reads/writes between images on the same node is a portable alternative to XPMEM since it has been supported in the Linux kernel since version 2.4 [7].

Finally, the new PGAS library does not require huge pages, which were used by the Cray PGAS library. While huge pages may be useful for performance of applications with large data sets and certain access patterns, we have encountered issues when running CGE where memory fragmentation resulted in insufficient huge pages being available causing Cray PGAS to initialization to fail. Removing the use of huge pages has removed these initialization failures and, as our benchmarks will show, has not resulted in a significant reduction in performance.

3.1.2 Communication and Synchronization

The new PGAS library leverages MPI for handling the process management interface (PMI) with the work load manager and process synchronization. The communication of one-sided messages are implemented using MPI windows [8] to enable remote memory access (RMA) between images. Using MPI provides portability through multiple performant implementations and simplifies the development effort required for the new PGAS library.

When using MPI for RMA, the remote reads and writes to the symmetric heap between images are implemented using the MPI window feature, which allows processes to specify a virtual address range available for RMA operations [8]. All images create their window at CGE startup and use *MPI_Win_lock()* to create an access epoch to the windows for remote images that remain active until CGE exits. When an operation in CGE results in a put/get, the library first determines if the remote memory involved is on the same physical node or not. If the memory is on the same physical node the put/get is simply performed as a *memcpy()*. However, if the memory is not on the same node, an *MPI_Put()* or *MPI_Get()* is issued and the process may immediately block if the given operation requested blocking semantics; otherwise the RMA operation is tracked and no blocking is done.

The other main feature provided by MPI is process synchronization. In Coarray-C++ terms this refers to a *sync_all()*, which requires all images to participate and ensures all outstanding puts/gets have completed at the source and destination. Coarray-C++ also includes an *atomic_image_fence()*, which ensures all outstanding puts/gets for a given image have completed. These are often used in CGE when overlapping communication and computation to increase independence between images by reducing complete synchronization. In the new PGAS library when using MPI RMA, the *atomic_image_fence()* translates into *MPI_Win_flush_local()* calls for any source/target image that the given image has an RMA operation outstanding with. The *MPI_Win_flush_local()* takes an argument that specifies the target rank and ensures all RMA operations with that given target have completed. A *sync_all()* call translates into an *atomic_image_fence()* followed by an *MPI_Barrier()*.

3.2 Simplified Coarray-C++

Once the new PGAS was created, a simplified version of the Coarray-C++ header file was created targeting the new PGAS library rather than the Cray PGAS. This is a significantly reduced version of the original Cray Coarray-C++ that removes features not used by CGE, such as *coatomic* and *cofuture* objects, for the sake of reducing development efforts and code maintenance. Since most of the implementation details are actually in the PGAS library and not the Coarray-C++ header file, the changes to the header file mostly consisted of translating calls from the Cray PGAS to the new PGAS. This includes calls for symmetric heap allocations/frees, image fence and barriers, and remote put/get calls.

The Cray Coarray-C++ also included more code for supporting collectives, such as sum/min/max, which have been simplified in the newer Coarray-C++ header file because the implementation has been pushed down into the PGAS library. The PGAS implementation still provides templated versions of these functions for offering type checking at compile time while also offering generic versions for a C based interface.

3.3 Optimizations

With the new PGAS library a couple optimizations were added based on how CGE often uses Coarray-C++. First, a new function, *coexchange()*, was added to the new Coarray-C++ header file with the necessary support also added to the new PGAS library. This function enables images to easily exchange simple data values they have for all other images. For example, often in CGE all images have start/count values for all other images that inform the images where to pull data from and how much data to pull. The prototype for *coexchange()* is shown in Listing 1.

```
template < typename T >
void coexchange( coarray<T []>& dest, coarray<T []>& src )
```

Listing 1. C++ prototype for coexchange

The *src* and *dest* arguments are both *num_images()* in length and upon entry the *src* provides the data for all

images it has data for such that *src[dest_image]* is the data for destination image *dest_image*. Upon completion, *dest* holds the received data for a given image such that *dest[src_image]* is the data received from *src(src_image)[this_image()]*. The exchanging of keys is done in a group-aggregated manner, similar to what is done for operations such as scan/join/merge since this has been shown to scale well [3]. The motivation for this new function will be explained below when discussing the MPI implementations available for the containerization of CGE.

The second optimization added was the ability to separate an image fence for puts and gets so that an image can fence on one type of remote operation while not fencing for the other. This meant the addition of two new functions to the Coarray-C++ header file, *atomic_image_put_fence()* and *atomic_image_get_fence()*, that perform image fences for outstanding RMA puts and gets, respectively. In the new PGAS library, outstanding put and get RMA operations for a given target image are tracked separately, which enables the flush operations to be done separately. This can be useful for CGE where communication and computation are often overlapped, such as with group-aggregated messaging [3], and the outstanding get and put operations typically involve different images. An example of where this is now used in CGE will be discussed more below.

4 DEPLOYMENT

To enable optimal execution on multiple platforms, CGE can be deployed either natively or in a container. The native deployment is the manner more typical for XC where CGE is available in a loadable module. The container deployment uses Singularity containers to more easily allow CGE to be executed on multiple platforms while minimizing the effort required for deployment. These two methods of deploying CGE will be discussed in more detail in the subsequent sections.

4.1 Native Build

On the Cray XC, CGE is released as a module that users can load, which makes the install process simple and allows users to easily switch between installed versions. We will refer to this deployment method as native because CGE is executed directly, via either srun or aprun, rather than through a container image. The new CGE that uses MPI and POSIX shared memory is installed in the same manner for XC systems and current plans are to deploy natively for Shasta (EX) systems.

A key motivation for native deployment on XC and Shasta systems is to enable CGE to build using the Cray MPICH libraries that are optimized for the given hardware. The Cray MPICH libraries should enable CGE to achieve the most optimal performance across the Cray hardware while also simplifying the deployment since it uses the existing mechanism for installing and users are familiar with using modules.

4.2 Containerization

Recent changes to CGE to create the new PGAS library and Coarray-C++ header were done to increase the number of platforms that CGE could efficiently execute upon, as well as enabling CGE to be deployed within a container. With containerization, CGE could be deployed on multiple platforms locally or in a cloud instance with all of the required packages included in the container to simplify deployment. Singularity was chosen for containerization since it is intended for HPC applications [9].

With Singularity containers there are generally two approaches to running MPI applications. The first method is called the hybrid model, which is where the MPI installed on the host system and the MPI within the container are both used to execute the application. The second model is the bind method in which there is no MPI within the container and the host MPI is bind-mounted into the container to be used by the application. The advantage of the hybrid model is that it is simpler, however, it requires the MPI inside the container to be appropriately built in order to get optimal performance. The bind method has the advantage that performance should match because the host MPI is being used, however it requires pulling host libraries into the container which could lead to problems such as version mismatches [9].

For the CGE Singularity container we chose the hybrid model to minimize the effort for porting the container to multiple platforms. In order to achieve optimal performance, the container includes the following components:

- Mellanox OpenFabric Enterprise Distribution (OFED) [10]
- Unified Communication X (UCX) [11] or Open Fabric Interfaces (OFI) [12]
- MPI (OpenMPI [13] or MPICH [14])

The OFED stack is a key component in order to achieve optimal performance as it is used to provide Infiniband $verbs$ [15] support for both UCX and OFI without requiring host libraries to be bind-mounted into the container. The components are installed in the container in the order they are listed with UCX/OFI and MPI configured to use the installed OFED software.

4.2.1 MPI Options

Given that MPICH and OpenMPI (OMPI) are both widely used open source MPI implementations but are not API compatible with one another, we chose to create different Singularity containers for CGE targeting each MPI. The first was built with MPICH using either OFI or UCX and the second was built using UCX and OpenMPI. This was done to enable the CGE container to execute well on the widest range of platforms. Each container version has different advantages and disadvantages, which will be discussed below.

One significant advantage for CGE when using MPICH in the container is the ease of execution using *srun* directly, which is possible due to the builtin support for PMI-2 within MPICH. Another potential advantage is that CGE built with MPICH would be API compatible with other MPICH based implementations, such as Cray MPICH, which could enable the bind-mounting of the host MPI into the container. However, given our focus on the hybrid model of the container this was considered a less important ability. When using MPICH with OFI we encountered OFI timeout errors

caused by all-to-all communications of small messages (i.e., < 32 bytes) in CGE when executing on 32+ nodes. This was the motivation for creating the new *coexchange()* which replaced several of these problematic messages in the query engine portion of CGE with more group-aggregated ones that avoided the OFI timeouts. However, while we updated some of CGE to use the *coexchange()* there are still multiple places in the build phase of CGE that encounter the OFI timeout at 32+ nodes, preventing our ability to build databases with MPICH+OFI at large node counts. These OFI timeouts led us to try MPICH with UCX, which improved resiliency at larger node counts and faster performance over using OFI, which will be discussed later in the benchmarks section.

The second MPI option for containerizing CGE is OpenMPI, which is similar to MPICH in that there are performant builds available on a large number of platforms. The default communication layer in OpenMPI versions 4.x and newer is UCX so we combined UCX and OpenMPI in the container. One potential advantage of OpenMPI for containerization is that the support for OpenMPI with Singularity may be better than the support for MPICH, at least based on the initial focus on OpenMPI by Singularity [9]. Another advantage that arose from our tests is the reliability of OpenMPI with UCX was better than that of MPICH and OFI based on our ability to successfully run more tests and larger scales with OpenMPI and UCX. This will be discussed in more details in the benchmarks section below. One disadvantage to OpenMPI is that it typically requires installing OpenMPI on the host machine in order to correctly launch the containers and handle the PMI interaction between the host and containers since the ability to be launched directly via *srun* is not enabled by default for OpenMPI. Further, our efforts to include *slurm* and PMI-2 inside the container with OpenMPI could successfully build but failed to create the RMA windows at CGE launch.

5 Performance Improvements

Multiple performance improvements have been made to CGE since the latest benchmarks in 2018. The changes focused on the idea of aggregating messages into larger messages to reduce the overall number of messages since this has been shown in previous studies to greatly improve performance and scalability [3].

5.1 Dictionary

The first significant performance optimization made was in adding the ability for the dictionary to fetch the strings for a given set of integer identifiers as large blocks. The dictionary is used to provide a mapping from the RDF string literals to the integer CGE uses internally to represent literals as part of quads [3]. Several operators often need to fetch a large number of dictionary strings, such as during a FILTER operation or when generating the query results to return to the user. Previously this meant multiple calls to the dictionary to fetch a single string at a time, which could limit scaling and performance due to the number of messages occurring simultaneously. To improve this, the dictionary now can take an array of integers and fetch all strings from a given source image as a single block using group-aggregated messaging. This requires all images to participate but significantly reduces the number of messages and, as will be shown below, can drastically improve performance.

5.2 Inferencer

During inferencing all quads are validated before inserting them into the database, which requires fetching the RDF strings for each field and verifying the given string is valid for the field. For example, this check verifies that string literals are not used as the subject or that blank nodes are used as a predicate. This was being done by fetching the RDF string for each field from the dictionary one at a time as needed when evaluating the new quads. The inferencer caches strings to reduce the number of fetches for common strings, however, the number of messages was still $O(num\_inferred\_quads)$ which could limit scalability due to the total number of concurrent messages. To help address this, the inferencer was updated to use the new block fetching feature of the dictionary to enable all strings that will be required for verification to be fetched as large blocks in a group-aggregated manner. This drastically reduced the inferencing time for larger datasets. For example, using 32 Broadwell nodes with 16 images per node for a total of 512 images, the inferencing step on the Lehigh University Benchmark 100K [16] infers $\sim$4.9B new quads and the total database has $\sim$18.2B quads. The inferencing step previously took $\sim$1797 seconds when fetching the strings individually and after updating to use block fetching the time was cut down to $\sim$159 seconds. This clearly demonstrates the significant performance improvements possible when trading larger messages to reduce the number of messages.

5.3 Query Engine

Given the desire to execute CGE on multiple platforms, including large shared memory machines such as the SDF, recent optimizations were made to the scan/join operators that are part of nearly all queries to better utilize a shared memory architecture. Much of the work involved with the scan/join operators is for tracking valid identifiers for the variables involved in a query. These bindings for variables are tracked for each scan/join operation in a query and the bindings from a given scan/join are used to filter out solutions in subsequent scans/joins. A significant portion of the time for getting and setting these variable bindings is spent marshalling the data into large blocks in order to perform group aggregated messaging [3] of the bindings. When executing on a shared memory machine such as the SDF, this data marshalling is unnecessary and just adds overhead so these operators were modified to detect when CGE is executing on a shared memory machine so this data movement can be skipped. Instead, when CGE is executing on a shared memory machine images will directly read-/write to each other's memory for handling the variable bindings.

Other key optimizations made recently to CGE also focused on the scan and join operators since they are part of nearly all basic graph pattern searches. Each of these operators tries reducing the number of potential solutions

by tracking the identifiers for the unbound variables encountered in one subset of solutions and using these to filter potential solutions from subsequent solution subsets. Previous work to enable CGE to efficiently load and query a trillion triples focused on improving the performance of tracking the bound variable identifiers in scan and join and drastically reduced the operator times as well as improved scaling [3]. That work focused primarily on changing the image-by-image aggregation of bound identifiers to the group-aggregated version so that each image would only share bound identifiers with one other image per group for a total of $O(num\_groups())$ messages rather than $O(num\_images())$ messages per image. The group-aggregated messaging is described in more detail, including with pseudo-code, in our previous study benchmarking CGE with a trillion triples [3].

Recent optimizations to scan and join again focused on improving the performance of tracking the variable bindings and using them to filter solutions. The first of these optimizations was to modify how bound identifiers are grouped per node. In the original group-aggregated messaging, each image works with a partner image on each group. The partner image is the image on the given group that is at the same relative offset. For example, image 0 would always partner with the lowest numbered image in a group so if there are 16 images per group, image 0 would partner with itself, image 16, 32, 48 and so on. For the variable bindings process, this aggregation results in a total of $O(num\_groups() * num\_images())$ messages since each image communicates with one image per group. The actual work to fetch and set the variable bindings overlaps the fetching of bindings from one image while processing the bindings from another. Even with the group-aggregation, the communication time still becomes a bottleneck for performance causing stalls in the loop over images. This is especially true for the process of fetching bindings to determine which solutions to filter because the potential solutions are pulled to the image that contains the bindings and a keep/discard status is sent back for each of those potential solutions to the source to signal which to eliminate.

The improvement made to the group-aggregation method was to change the images each image partners with so that they work with consecutive images rather than images at even strides. Now each image holds binding data for $num\_groups()$ images in a dense range. For example, image 0 holds the data from its group members for the first $num\_groups()$ images. If there are 16 images per node and 512 total images, image 0 holds the data for images 0-31 and image 15 holds the data for images 480-511. This new aggregation means each image only needs to communicate with $num\_groups()/group\_size()$ images to send all of the identifiers they hold for their group members. Further, as more images are added per node the overall number of messages per image goes down since the number of messages is divided by the $group\_size()$. In the original version of group-aggregation, the number of messages each image was responsible for was always $O(num\_groups())$ but now it is $O(num\_groups()/group\_size())$. To evenly distribute the messages for a given destination group across the images in the group, teams are formed out of all images that communicate with a given group. The images offset within this team is then used to choose a partner image on the given destination group in a round-robin manner.

The second optimization made to the variable bindings step was to reduce the size of the messages both sent for the identifiers as well as the 1/0 status flags returned to the source. Originally, the full 64-bit identifier was sent to each destination image but now only a 32-bit index is sent which represents the offset into the bit-vector used to track bound identifiers. Additionally, the 1/0 status flags previously were sent as full words and reused the source image's identifier buffer to receive the status words. This has now been modified to create a new bit-vector to receive the status flags so that only a single bit per solution is sent back to the source.

One final optimization was made to the variable bindings process that leverages a new feature added to the CGE PGAS library, which is the ability for images to fence on puts and gets separately. As mentioned above, each image loops over the set of partner images to fetch the appropriate bindings and return the status bits, if needed. Double-buffering is used in this loop so that the images can begin fetching the next set of indices while processing the current set. After processing the current set of indices the image needs to do an *atomic_image_fence()* to ensure the fetching of the next set of indices has completed. However, in the case of returning the 1/0 status bits this resulted in the image not only fencing for the get to complete but also the put that was issued to return the current status bits. With the ability to separate these fences, the variable bindings loop will issue the put for the status bits and then fence on just the outstanding get for the next set of indices. Once an image has completed processing the variable bindings from all of their partners they will ensure the puts have all completed as well. This decoupling of the put and get fences helps prevent stalls in the computation portion of the loop that could happen with the immediate fence on the put operation.

To test the combined effects of these optimizations on the variable bindings step in scan and join, LUBM100K was used with query 9 running on 32 Skylake nodes with 16 images per node. Query 9 is a key benchmark query because it performs multiple complex joins in order to search the entire graph for a given triangular relationship [16]. Prior to these optimizations, the query 9 time was 8.6 seconds and after the optimizations the time was reduced to ∼7.0 seconds, clearly demonstrating the advantage of reducing the number of messages and improving the overlap of gets/puts.

6 BENCHMARKS

To demonstrate performant execution across multiple architectures we present timing results for both the database build and LUBM query 9 performance across multiple system configurations. Query execution time reported is the strict query time and does not include the time required for writing the results to the file system which is common practice. Due to resource constraints, we were not able to run the same size LUBM dataset across all the different system configurations, but selected a database size sufficient

to utilize most of the memory available on the smaller system configurations (SuperDome Flex and AWS parallel cluster).

6.1 System Configurations

Below is a brief description of each of the system configurations used for benchmarking.

6.1.1 Cray XC

The Cray XC results were run on an internal 370 node XC-50 system development system utilizing the Aries interconnect. This XC system contains a mixture of compute node types (Intel Broadwell, Intel Skylake, Intel Cascade Lake, and ARM processors). Benchmarks were primarily run on a mixture of dual-socket 48-core Skylake and Cascade Lake nodes, ranging in frequency from 2.1-2.4GHz. The majority of these nodes have 192 GB DDR4-2666 memory but 63 of the Cascade Lake nodes have the larger 384 GB DDR4-2933 memory. The attached Lustre file system is a Sonexion CS-L300N system with 8 OSTs providing 655 TB of storage. Database build and load times are dominated by I/O performance to/from the Lustre filesystem so I/O system performance is an important consideration.

6.1.2 Cray CS

The Cray CS results were run on an internal CS500 development system utilizing HDRI InfiniBand (using Mellanox ConnectX-6 HCA). The CS system also contains a mixture of compute node types including both Intel Cascade Lake (dual-socket nodes, 44-cores per socket, 2.4GHz, 192 GB DDR4-2934 memory) and AMD EPYC Rome processors (dual-socket nodes, 64-cores per socket, 2.0 GHz, 256 GB DDR4-3200 memory). Benchmark results using 128 nodes were run across both the Cascade Lake and the Rome processors. For the Cascade Lake nodes, it also uses a daughter card to enable Mellanox's Socket Direct. The attached Lustre files system is a Lustre Sonexion E1000 with 10 OSTs providing 2.9 PB of storage.

6.1.3 HPE Cray EX

The HPE Cray EX results were run on a 1024-node internal Shasta development system using Slingshot 10 interconnect. The network topology for this development system consisted of 8 groups, with 16 switches per group, and 128 nodes connected at each group and each node in the system hosts dual Mellanox ConnectX-5 Network Interface Cards. Compute nodes in this system are using AMD EPYC Rome compute processors (dual-socket nodes, 64-cores per socket, 256 GB DDR4-3200 memory) ranging in frequency from 2.0-2.25GHz. The attached Lustre file system is a Clusterstor system with 12 OSTs providing 2 PB of storage.

6.1.4 AWS Parallel Cluster

For the AWS deployment we used the AWS ParallelCluster open source cluster management tool to configure and deploy a 32 node HPC cluster. We used compute resources from the us-east-1 region, selecting instance type c5n.18xlarge for the compute partition, since this instance types supports the Elastic Fabric Adapter (EFA). The cluster was configured to use the SLURM workload manager and also included an attached Amazon FSx Lustre file system. The cluster comes with OpenMPI targeting OFI with efa enabled. Since our intent was to use AWS to test cloud deployment, we also installed Singularity on the system.

6.1.5 HPE SuperDome Flex

Benchmarks were run on an 8-socket HPE SuperDome Flex system with 3 TB memory (DDR4-2933). Each 2.9 GHz Intel Cascade Lake socket has 24 cores for a system total of 192 cores.

6.2 Database Build

The database build process primarily involves parsing one or more N-Triple [5] or N-Quad [17] files, storing the unique RDF literal strings in the dictionary and creating a mapping between unique integer identifiers (HURI) and using these identifiers to create the integer quads stored in the quads table. The build step sorts all of the unique RDF strings after processing all input files and update the HURIs based on the sort order. This is done to enable optimizations in FILTER and other operators that compare literals using the HURIs rather than by the RDF strings. Finally, inferencing is executed on the loaded triples/quads if the user provided a set of rules. After building a database, a checkpoint is done to save the internal representation to disk to enable faster reloads of the same database. More information on the build process can be found in our previous study [3]. Figure 1 shows the flow of the steps involved in the build process.

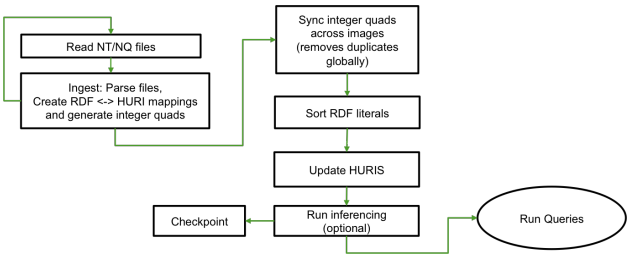

Fig. 1. CGE Database Build Flow Chart

The build process was benchmarked on the Cray CS, XC and EX systems using LUBM200K, which is ∼4.59 TB of raw N-Triples data on disk with ∼26.7 billion quads before inferencing and ∼36.4 billion quads after inferencing. On the AWS parallel cluster instance we used LUBM100K, which is ∼2.29 TB of raw N-Triples data on disk with ∼13.3 billion quads before inferencing and ∼18.2 billion quads after inferencing. Finally, on the SuperDome Flex we used LUBM25K, which is ∼583 GB of raw N-Triples data on disk with ∼3.3 billion quads before inferencing and ∼4.6 billion quads after inferencing. We used smaller sizes of LUBM on AWS and SDF because of the resources we had available. As discussed above, the AWS cluster was only 32 nodes and the SDF was only 8 sockets.

Table 1 below shows the times for the different build steps of LUBM200K on CS, XC and EX when running on 32 nodes with 16 images per node. As the numbers show, the time for reading the RDF triples from disk is a significant portion of the total build time. The remaining steps in the

build phase is what we refer to as the "build processing" since it is the work to parse and convert the input to the internal format once it is loaded from disk.

TABLE 1
Times in seconds for Build Steps for LUBM200K Using 512 Images

Build Step	CS MPICH+OFI Singularity	CS MPICH+UCX Singularity	CS OMPI+UCX Singularity	XC Cray MPICH Native	EX Cray MPICH Native
Read RDF	658.2	659.8	653.9	414.4	551.9
Ingest Quads	253.5	239.5	214.0	236.6	279.6
Sync Quads	39.6	32.3	27.7	31.4	35.8
Sort Literals	76.3	74.3	59.9	73.4	90.2
Update HURIs	295.7	211.2	174.6	207.0	225.2
Inference	663.6	539.2	458.9	547.2	553.0
Total	1986.9	1756.3	1589.0	1510	1735.7

Figure 2 shows the scaling of the build processing time for LUBM200K when executing on 32, 64 and 128 nodes with 16 images per node. The build process scales fairly well when using OMPI on the cluster as well as when running natively on XC and EX. The MPICH performance on the cluster using UCX scales reasonably as well up to 64 nodes but is unable to complete at 128 nodes. The database build failed when using MPICH+UCX at 128 nodes because buffers were not correctly being transmitted which caused CGE assertions to fail when it detected invalid input values in the receive buffers. Further, when using MPICH+OFI, the build process fails when using more than 32 nodes because CGE encountered OFI timeout errors caused by too many ranks doing put/get operations to the same target rank.

On AWS we leveraged the OMPI+UCX Singularity container and we used LUBM100K since we had limited lustre space and only had a 32 node parallel cluster. The LUBM100k build was performed using 32 nodes and 16 images per node (i.e., 512 images) and the processing portion took 693.8 seconds and the reading of the raw N-Triples files took 2109.9 seconds. The build processing time translates to ∼51,300 quads/image processed per second which compares to ∼76,123 quads/image processed per second when using the OMPI Singularity container on the CS cluster.

For the SDF we used the MPICH+OFI based Singularity container. The SDF was only an 8 socket machine and did not have a parallel file system so we used the smaller LUBM25K for our tests. Given that no parallel file system was available the build time was dominated by the time to read the RDF, which took 981.7 seconds for the 583 GB of raw data. The build processing time took 873.5 seconds when using 24 cores per socket for a total of 192 images, which equates to 27,167 quads/image processed per second. Note that much of CGE has not yet been optimized for running on a large shared memory machine so considerable time is spent marshalling data unnecessarily. Once more of CGE is updated to reduce this data movement the build performance on the SDF should improve. Also, the build process did not ensure optimal NUMA placement to best utilize all sockets, which could certainly impact performance.

6.3 Query Performance

Figure 3 shows the scaling of query 9 for LUBM200K when executing on 32, 64 and 128 nodes with 32 images per node for XC, CS and EX. On the XC we tested CGE using the

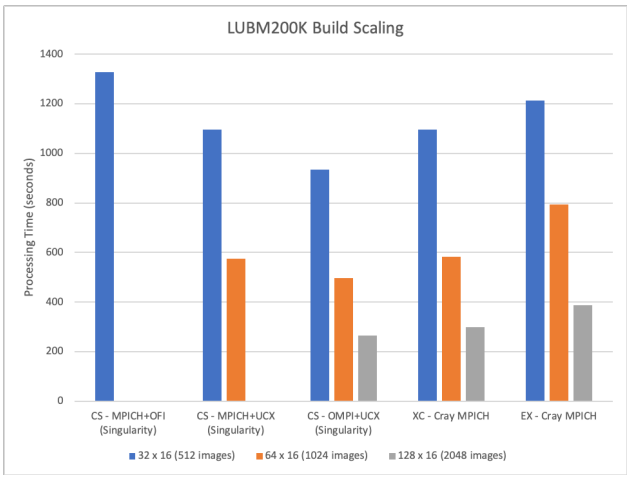

Fig. 2. CGE LUBM200K Build Scaling

Cray PGAS as well as with the new PGAS library based on MPI to look for any significant performance differences. As the numbers show, there is no difference in performance when using MPI rather than the Cray PGAS on the XC. Further, as previously mentioned the new PGAS library no longer requires the use of huge pages. This is an important improvement for CGE because requiring huge pages has caused problems at CGE startup on nodes with insufficient huge pages.

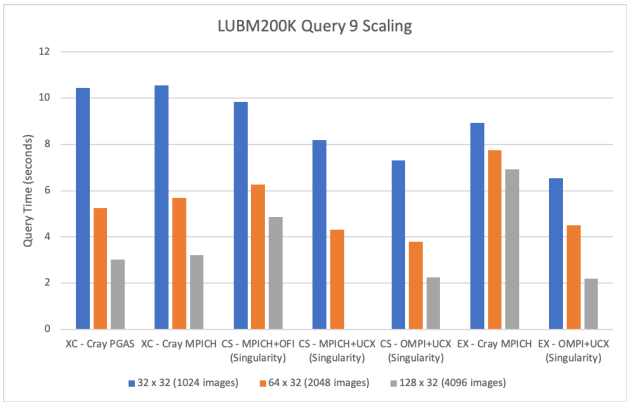

Fig. 3. CGE LUBM200K Query 9 Scaling

On the CS we tested all three of our container versions: MPICH+OFI, MPICH+UCX and OMPI+UCX. With the changes to scan/join to leverage the new *coexchange()* the MPICH+OFI was able to run up to 128 nodes though there was limited scaling beyond 64 nodes. The MPICH+UCX again performed better than MPICH+OFI, however, it was unable to execute properly at 128 nodes due to errors in the buffers being transmitted. Leveraging OMPI+UCX was the best performing container for query 9, just as it was for the build processing.

On the EX we were able to test the native compilation and execution of CGE using Cray MPICH as well as the OMPI+UCX singularity container. As the query results show in Figure 3, using Cray MPICH had scaling limitations even at 64 nodes and performed considerably slower than CGE inside the OMPI+UCX container. We have not yet

determined why the native build performed so poorly compared to the containerized version and this is an ongoing investigation. The ability to run the same OMPI+UCX container on EX as on the CS and AWS demonstrates the ease of portability offered by containerization. Finally, given the fact that the OMPI+UCX singularity containerized version of CGE performed the best across all platforms clearly illustrates that CGE can now run optimally across multiple architectures while minimizing development effort.

7 Conclusions

In this paper, we have demonstrated how we were able to containerize CGE in order to enable easy porting to multiple platforms. We discussed the work required to enable the containerization, which mainly focused on removing dependencies on the Cray Coarray-C++ and PGAS library originally used by CGE. To do this, a new PGAS library was created based upon MPI RMA operations and POSIX shared memory and a simplified Coarray-C++ was created to utilize the new PGAS. The MPI and POSIX shared memory standards were chosen for multiple reasons, including greater portability, ease of containerization and performance on multiple platforms. We also discussed the components required within the container to enable optimal performance of CGE across platforms.

We also discussed recent optimizations made to CGE to improve some operations, including scan, join and inferencing, and why these improvements were more important now that CGE was executing on architectures other than XC. Through our benchmarks we were able to demonstrate the performance improvements gained from these changes, which were significant in cases such as inferencing where the fetching of dictionary strings was modified to use group-aggregated messaging.

Finally, we have shown the performance and scaling of CGE on several architectures, including XC, CS, EX, SDF and AWS, when running either natively or inside a container. We demonstrated the ease of executing CGE on multiple platforms by leveraging the same container on multiple architectures and verified that CGE can run optimally on multiple platforms by comparing the performance and scaling on all architectures to the performance of CGE on XC using the original Cray PGAS.

8 Future Work

As discussed in Section 6.2, the MPICH+OFI containerized version of CGE suffered from OFI timeout errors that prevented scalability and limited performance. We plan to do further investigation into these timeouts to see what the cause(s) may be and how they can be avoided. As part of this effort, we may consider going directly to OFI rather than using MPI for the RMA operations to see if that avoids or at least reduces the timeout failures. Further, since using UCX for both MPICH and OMPI provided the best performance, we plan to investigate whether we could interface with UCX directly rather than MPI for the RMA operations. By calling UCX directly the container could use MPICH and UCX to ease the launch of the CGE container directly from $slurm$ while avoiding the OFI timeouts and gaining the performance improvements of UCX.

Finally, as discussed in Section 6.3, there is a significant performance difference on the EX when executing CGE natively using Cray MPICH versus running the containerized CGE built upon OMPI+UCX. Further investigation into the cause of the performance difference and limited scaling is planned to determine why there is such a difference and if any changes could be made to CGE and/or Cray MPICH to improve the native performance.

References

[1] K. Maschhoff, R. Vesse, and J. Maltby, "Porting the Urika-GD graph analytic database to the XC30/40 platform," in *Cray User Group Conference (CUG '15)*, Chicago, IL, 2015.

[2] K. Maschhoff, R. Vesse, S. Sukumar, M. Ringenburg, and J. Maltby, "Quantifying Performance of CGE: A Unified Scalable Pattern Mining and Search System," in *Cray User Group Conference (CUG '17)*, Seattle, WA, 2017.

[3] C. Rickett, U. Haus, J. Maltby, and K. Maschhoff, "Loading and Querying a Trillion RDF Triples with Cray Graph Engine on the Cray XC," in *Cray User Group Conference (CUG '18)*, Stockholm, Sweden, 2018.

[4] T. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, 2013.

[5] D. Beckett, "RDF 1.1 n-triples," W3C, W3C Recommendation, Feb. 2014, https://www.w3.org/TR/n-triples/.

[6] "TCMalloc," https://github.com/google/tcmalloc, 2021.

[7] "shm_overview(7) — linux manual page," https://www.man7.org/linux/man-pages/man7/shm_overview.7.html, 2021.

[8] "MPI Forum," https://www.mpi-forum.org/, 2021.

[9] "SingularityCE User Guide," https://sylabs.io/guides/3.8/user-guide/, 2021.

[10] "Mellanox OpenFabrics Enterprise Distribution for Linux (MLNX_OFED)." [Online]. Available: https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed

[11] "Unified communication x." [Online]. Available: https://www.openucx.org/

[12] "libfabric." [Online]. Available: https://github.com/ofiwg/libfabric

[13] "Open MPI: Open Source High Performance Computing," 2021. [Online]. Available: https://www.open-mpi.org

[14] "MPICH: High-Performance Portable MPI," 2021. [Online]. Available: https://www.mpich.org/

[15] "Userspace verbs access." [Online]. Available: https://www.kernel.org/doc/html/latest/infiniband/user_verbs.html

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

[17] "RDF 1.1 n-quads," W3C, W3C Recommendation, Feb. 2014, https://www.w3.org/TR/n-quads/.