Quantifying Performance of CGE: A Unified Scalable Pattern Mining and Search System

Kristyn J. Maschhoff, Robert Vesse, Sreenivas R. Sukumar, Michael F. Ringenburg, James Maltby
Analytics R&D
Cray Inc.
901 Fifth Avenue, Suite 1000
{kristyn,rvesse,ssukumar,mikeri,jmaltby}@cray.com

Abstract—The Cray Graph Engine (CGE) was developed as one of the first applications to embody the vision of an analytics ecosystem that can be run on multiple Cray platforms. This paper presents CGE as a solution that addresses the need for a unified, ad-hoc, subject-matter driven graph-pattern search and linear-algebraic graph mining system. We demonstrate that the CGE implemented using the PGAS parallel programming model performs better than most off-the-shelf graph query engines with ad-hoc pattern search while also enabling the study of graph-theoretic spectral properties in runtimes comparable to optimized graph-analysis libraries. Currently CGE is provided with the Cray® Urika®-GX system and can also run on the Cray® XC40™ systems. Through experiments, we show that compared to other state-of-the-art tools, CGE offers strong scaling on graphs that are three orders of magnitude larger and more complex (long diameters, hypergraphs, etc.) while enabling computationally intensive pattern searches on those graphs.

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

1 Introduction

Pattern discovery and predictive modeling from seemingly related "Big Data" represented as massive, ad-hoc, heterogeneous networks (e.g. extremely large graphs with complex, possibly unknown structure) is an outstanding problem. While some of these challenges have been addressed in academic literature, and with off-the-shelf tools offering query engines for graph data, the academic contributions need benchmark and validation at scale before they can be adopted for real-world use cases in both government and industry. With graphs, the two worlds of pattern search with ad-hoc queries and pattern mining using spectral graph theory (e.g. shortest path, PageRank [1], betweenness centrality) have always been considered as two different aspects of graph analytics. While ad-hoc queries help understand vertex-centric sub-graph patterns, the spectral theoretic approach helps understand the global properties and behaviors of entities modeled into the graph.

Very few off-the-shelf compute and data architecture solutions are capable of handling petabyte scale graphs and can also operate with latencies to support interactive analytics. While some databases can offer interactivity with query mechanisms, computing spectral properties on graphs may not be supported. This capability to discover insights using subject-matter-expert intuition and the ability to use mathematical rigor to hypothesize or mine potential patterns of interest is emerging as a key use case within graph analytics. To the best of our knowledge, there is no unified system that is capable of both.

In this paper, we present Cray Graph Engine (CGE) built on parallel processing and distributed querying fundamentals as a potential solution that addresses this need. We present the design, implementation and benchmarking results of core kernels for high performance graph analytics within CGE and describe the performance characteristics of CGE on current and emerging Cray architectures. We show CGE is capable of: (i) speeding-up ad-hoc searches and graph-theoretic mining, and (ii) scaling to massive data sizes. We demonstrate CGE as a unified platform capable of handling use-cases from both worlds. Thus setting the foundation for temporal, streaming and snapshot analysis of massive graphs in future.

We start by discussing some of the trade-offs involved in architecting a multi-platform application. In the past [2], we have focused primarily on the software architecture targeting a single hardware platform. With CGE now running across multiple platforms, we describe how differences in hardware architecture, both at node level and network level, affect performance. There are also differences in the resource scheduling and job launch mechanisms used across the platforms which we refer to collectively as the workload manager (WLM).

We describe how we abstract away from the underlying WLM to provide a consistent application launch experience across platforms. Although this presents a user-friendly experience across platforms, and multiple WLMs on those platforms, it also creates challenges. The abstraction layer needs to ensure that when the user requests an application topology that the resources topology they receive from the WLM is appropriate and consistent.

We used the industry standard synthetic benchmark LUBM [3] that is a commonly used benchmark for pattern search and query. We demonstrate the utility and capability of CGE on several real-world datasets from the Stanford Network Analysis Project (SNAP) [4], using a variety of whole graph analysis techniques such as PageRank [1] and degree distribution. These graphs are expected to be quite challenging compared to synthetic benchmarks. We further extend our testing of query performance and scaling to a widely used real world life science database, the Uniprot [5] database of protein data and annotations.

The results from the use cases showcase the key capabilities of CGE as a unified platform for graph analytics. It allows accessing tuned native graph algorithms within a general purpose query language. This permits scale-up and speed-up of the most complex parts of graph analysis while still enabling flexible construction of domain-specific algorithms using the query language. Thus allowing users to unleash graph-theoretic mining on datasets that are three orders in magnitude the size of what workstations are able to handle.

The benchmarking results and comparisons allow us to demonstrate the scalability and performance benefits of CGE. This allows users to understand the kinds of workloads for which CGE would be beneficial and how performance differs across platforms. Users can then make a more informed decision about whether CGE is appropriate to their problem and on what platform it is best run.

2 Background

Our goal is to study graph analysis tasks under two broad categories: graph pattern matching and graph mining. Given a graph *G* and a pattern *P* that specifies the structural requirements, graph pattern matching retrieves all sub-graphs that satisfy *P* from *G*. This operation can answer questions like - "Who are all the patients who have received therapy *t1* and *t2* by physicians *p1* and *p2* respectively?". Graph pattern matching has been applied to studies in protein-protein interaction [6], social network analysis [7], fraud detection [8] and so on. On traditional/relational databases, the ability to search for ad-hoc patterns of interest would require complicated join-operations. The graph as a data structure avoids such complexity. A typical pattern matching workflow can be abstractly described as follows. Patterns (that represent a particular domain-specific hypothesis) are formulated by subject-matter-experts based on their domain intuition. The patterns are then expressed into queries and those queries are sent to and processed by graph analysis systems. Based on the analysis of retrieved sub-graph instances, experts may refine the results and continue the exploratory analysis. The extreme scale graph pattern matching problem is the sub-graph isomorphism problem (used for emergent pattern identification). The sub-graph isomorphism problem is computationally expensive and known to be NP-complete in the general case. Our working definition of graph pattern search is a manageable subset of the sub-graph isomorphism problem called the basic graph pattern (BGP) search.

The second category of problems in graph analysis is graph mining which aims to discover knowledge from graphs using mathematical properties drawn from graph-theory. Unlike graph pattern matching, graph mining does not begin with user-defined queries. Instead, these techniques are used to filter through data in exploratory ways by ranking/scoring associations. Graph mining techniques answer questions like "who is the most influential person in the social network?" (knowledge discovery) or "who is likely to be a friend of person *p1*?" (prediction). Well-known graph mining operations include graph-theoretic definitions of degree distribution, triangle counts, eccentricity, connected components, PageRank and so on. These mathematically inspired heuristics, as opposed to the ad-hoc intuition driven with graph pattern matching, complements the pattern search capability in study of behaviors within social networks [9], the resilience and stability of electrical grids [10], web search engines [11], recommendation engines [12], etc.

Recent surveys on managing and mining graph data [13] and graph algorithms [14], [15] along with the study of laws and generating models [16], has helped design of tools such as SUBDUE [17], gSpan [18] , OddBall [19], Pegasus [20], NetworkX [21], GraphLab [22], [23] etc. for graph mining. However, not all of the algorithms and tools are able to scale up and handle massive datasets (in the order of terabytes). Pegasus or GraphLab, which can be instantiated on high performance cloud infrastructures, are restricted to mining homogenous graphs. On the other hand, graph-databases such as Neo4j [24], Titan [25], Trinity [26] etc. can host and retrieve massive heterogeneous graphs on commodity hardware, but do not have the data-mining functionalities of Pegasus or GraphLab.

A few major differences among graph analysis systems include system architectures (e.g. standalone, distributed, appliance), graph data models (e.g. RDF, property graph), graph data formats (e.g. N-triple, RDF/XML, GraphML), and query interface (e.g. query languages, APIs).

There have been several efforts towards general purpose graph processing. Such systems provide a computation model or a library/API that can be exploited to implement graph mining algorithms or graph pattern matching-based applications.

Triple stores (e.g. Jena [27], Sesame [28], RDFSuite [29], SPARQLVerse [30], EAGRE [31], TriAD [32], 4Store [33], YARS2 [34]) are a popular class of database for the storage and retrieval of RDF triples. They are a class of systems that are specifically designed for optimally storing, retrieving and querying graph data [35]. What triple stores have in common is that they focus on the storage and retrieval of RDF triples using SPARQL [36]. Another major class of graph databases instead use the property graph model including Neo4j, DEX [37], Titan, etc. These graph databases differ from each other, and Triple stores, in terms of the query language they support (e.g. Cypher [35], Gremlin [37], SNQL [38], etc.). In particular the lack of a commonly adopted standardized query language creates vendor lock-in and reduces portability of workloads.

NetworkX is a Python library for the creation, manipulation, and analysis of complex networks. Pegasus [20] is a Map-Reduce based implementation of graph algorithms that runs on Hadoop [39]. GraphX [40] is a graph processing system on top of Apache Spark™ [41]. The data models used by these systems are quite diverse. Pegasus uses adjacency matrices, but NetworkX and GraphX use the property graph as their data models. As these systems emphasize efficient processing of graph mining algorithms, they do not provide any query language processing capability. GraphX and NetworkX provides a set of APIs that can be used for graph pattern matching, but they require programming efforts to perform graph pattern matching.

Based on our survey of infrastructures, tools and algorithms, we identify the following scientific and technical challenges for graph-mining at scale:

(i) There are no reliable and scalable solutions and tools for integrating, storing, retrieving and processing massive heterogeneous graph-datasets; (ii) Properties of large, real-world, ad-hoc, heterogeneous graphs (allowing different types of vertices and edges) are not as well understood as those of homogeneous complex networks [16], [42]. Existing methods assume apriori knowledge of a well-understood model for network formation (e.g. Barabasi-Albert [9], or Erdos-Renyi [43]) (iii) Analysts that work with massively parallel processing databases have difficulty with even relatively simple graph-theoretical analysis. Primarily because of the extreme difficulty in writing (SQL) queries for graph patterns that are not known in advance, involve many vertices, and require approximate matching (qualities which all co-exist in graph-mining applications) [44]. Further Graph-theoretic feature identification via interactive querying algorithms on distributed storage solutions (off-the-shelf IT hardware and software) usually requires complex implementations. These limit flexibility, assumes sparse relationships between entities in the graph and often fails to scale to the size of real-world problems.

3 Architecture

In our previous paper [2] we discussed the architecture of Cray Graph Engine (CGE) primarily from a software perspective. As we have continued to develop the application into a true multi-platform application we have had to become more aware of how the differing hardware architectures influence our performance. Taking different architectures into account has caused us to make subsequent design decisions with those in mind. In this section we first detail the practical differences between the two hardware architectures we support - XC and Urika-GX - before discussing how they influence both observed performance and our design.

3.1 Hardware Differences

The primary difference between the two platforms is the physical network. While both platforms use the Cray Aries network, the way in which the two platforms are connected is very different. In the case of the XC platform [45], the Aries chips are connected directly to the processors on the Cray proprietary compute blades. However, on the Urika-GX platform, which uses commodity Intel motherboards, the Aries chips are located on separate Dual-Aries Network Cards (DANCs) which are interfaced with the motherboards via PCIe connections. This introduces an extra physical hop in the network routing which is not present on the XC platform. This typically manifests as an additional latency in communications which leads to a measurable slowdown in throughput relative to running on the XC platform.

As can be seen in Figure 1, there is a large gap in TCP/IP performance of around 20 GiB/s. For non-HPC applications

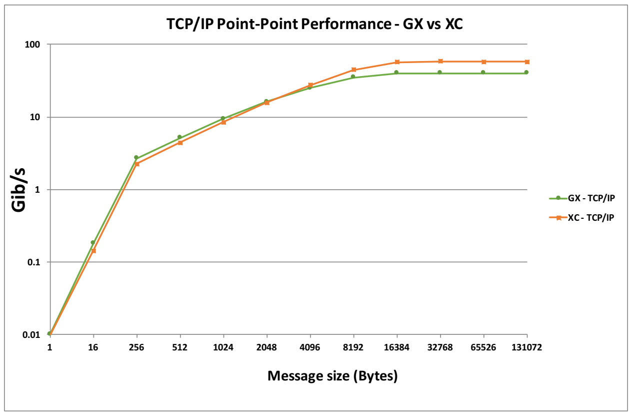

Fig. 1. TCP/IP Point-Point Performance Comparison - GX vs XC

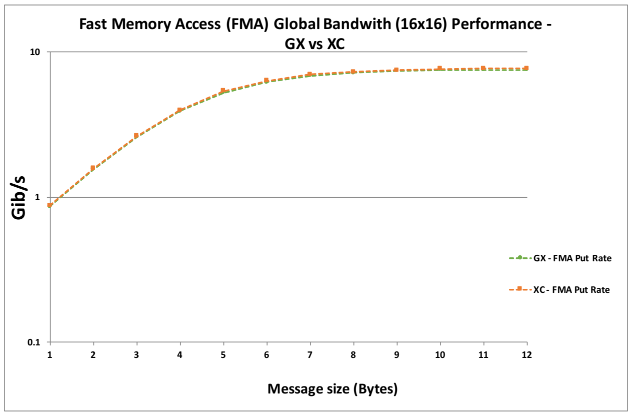

Fig. 2. FMA Global Bandwith Performance Comparison - GX vs XC

like Apache Spark which rely upon TCP/IP for communications this can make a difference across platforms. Although we would note that the throughput seen is typically far larger than these applications would usually expect to see on commodity hardware that would usually rely on 1 or 10 Gig Ethernet.

Fast Memory Access (FMA) is a feature of the Aries network that permits efficient communication to and from remote memory. As can be seen in Figure 2, FMA performance is essentially identical between platforms. The GX is marginally slower but only by a few MiB/s which means actual performance should be relatively close as we will see later in Figure 5. Therefore HPC applications like CGE, which primarily relies upon FMA traffic, should still be able to achieve similar throughput across the two platforms. We would note that there is a measurable latency of approximately 240 nanoseconds on the GX platform as compared to the XC platform. Although this is a tiny amount, over the lifetime application we may make hundreds of thousands of network requests, and as such this latency will add up. Therefore as we will discuss in Section 3.2 we need to structure our communications to take this into account.

On the other hand, the Urika-GX architecture has some advantages that the XC does not. A typical GX node will have much larger local memory than an equivalent XC node. Additionally, the presence of local SSD storage allows us the option of leveraging that as additional scratch space if we wish. While we do not do this currently we are actively exploring this as a future enhancement. Primarily, this will provide a way to achieve a better parity of scaling between the two platforms since on the GX platform we have an upper cap on the number of physical nodes that limits our scale on that platform.

As we will show later in Section 4, the observed performance difference between Urika-GX and XC is minimal. However, we would note that the performance difference was not always this close. As we will discuss in this section, a lot of thought and work has gone into achieving this performance parity between the two platforms.

3.2 Influences on Design

The differences in the network architecture has had a clear influence on our software architecture and design decisions. Primarily this has focused on finding ways to exploit locality as much as possible to avoid unnecessary network traffic.

One interesting example of this is the implementation of the SPARQL [36] `ORDER BY` operator which orders query results according to one or more ordering conditions. In general, we implement this by having each image first sort its local set of solutions according to the ordering conditions, followed by a global sort where images exchange solutions in order to arrive at the final sort order. Depending on the amount of solutions to be sorted this can be a very expensive operation involving a large volumes of network traffic. In order to avoid congestion in the network we explicitly schedule the exchanges of data to avoid all to all communication, instead using a staggered data exchange where images perform pairwise exchanges in a sequential fashion. Additionally our implementation of global sort is a variant of merge sort implemented such that most merges occur on node, therefore most data exchange is done via the shared memory on the node avoiding the network entirely.

However, in cases where the set of solutions to be sorted is relatively small the expense of the network exchange can far outweigh the cost of actually sorting the data. In these cases we instead choose to do the entire sort locally on a single image. This still requires some degree of network data exchange in order to gather the relevant data on a single image, but this is a much simpler network operation than that required to complete a global sort. In some of our standard test cases we were able to see an order of magnitude improvement in the sort time when the sort was small enough.

Another example pertains to load balancing. Ideally we want each image to perform an equal amount of work such that images minimize the time spent waiting at synchronization barriers and thus CPU cycles wasted. However, by its very nature of being a database engine images naturally become unbalanced over time in the course of answering queries. For example, the SPARQL `FILTER` operator eliminates solutions where an expression does not evaluate to true. It is possible that a given image might eliminate all of its solutions while another may eliminate none. Therefore despite the costs of using the network we have found that it is better for overall performance to periodically re-balance the intermediate solutions across all the images. This ensures that each image has roughly the same amount of work to do for any step of answering the query and allows us to maximize the utilization of our compute resources.

Even our re-balancing algorithm is designed to minimize network traffic when possible. Whenever we can we first re-balance locally within each node, i.e. images running on the same physical node re-balance their portion of the work between themselves. This can be carried out entirely using the shared memory avoiding the network entirely. The current statistical distribution of solutions is then used to determine how unbalanced the load currently is across the entire application. We consider the work to be unbalanced if a given image has 10% more work to do than the global average amount of work. If this is the case then we proceed to do a global re-balance which does involve network traffic. However, since by this point we have already rebalanced locally the global balance should already be reasonable as a local re-balancing should have removed the extremes of the work distribution. Therefore most of the time we can avoid the global re-balancing entirely and the only network data exchange that happens is the exchange of statistics.

Another design decision that we took to maximize network performance was to structure our network usage to use non-blocking constructs wherever large data exchanges are necessary. By using non-blocking constructs in combination with multi-threading we are able to issue many parallel network requests against remote memory at the same time. In doing this, we are able to offset the additional latency encountered on some platforms by experiencing that latency in parallel and continuing to do work while waiting for requests to complete. The downside of non-blocking constructs is that they require us to be more careful about global synchronization. At some point we typically have to wait for any outstanding requests to complete in order for us to proceed with further work, as otherwise we may access uninitialized memory that leads to a variety of cryptic error conditions. Therefore there is a balancing act, we prefer non-blocking constructs for larger data exchanges but rely upon blocking constructs for simpler exchanges.

One other key influence on design and implementation was our widespread adoption of configuration settings. These can be used to tweak many of the parameters that may be sensitive to the platform upon which we are running. For example, the aforementioned threshold at which we decide to do a single image sort is a configurable setting. There are a variety of settings that exist most of which end users need never be aware of, nor should they ever need to change, since the default values for each of these has been arrived at through our own internal performance testing. Allowing these settings to be changed if necessary allows our support organization to provide customers with ways to tweak behavior in the field should they encounter a problematic query. Receiving this feedback from the field has allowed us, where appropriate, to tweak the defaults to better reflect real world use cases that our own internal datasets may not yet reflect thus enhancing the product for all users in the future.

3.3 Abstracting the Work Load Manager

One key decision that we made early on was that we wished to abstract away from the Work Load Managers (WLMs) used on the different platforms in order to provide our users with a consistent application launch experience. Rather than having to provide users with detailed instructions on how to launch our application on each platform we instead provide a wrapper that handles this for our users called `cge-launch`. Users provide application-specific options and a general specification of the desired runtime topology. Runtime topology is defined in terms of the number of physical nodes and is the number of images i.e. processes to run per physical node. Typically we expressed this in the form $N \times I$ Where N is the number of nodes and I is the number of images per node.

Our launcher then has the task of taking this desired runtime topology and translating it into a physical topology using the platforms work load manager. It uses information present in the users environment to determine what the underlying platform workload manager is and then to generate an appropriate translation to the suitable job submission commands, e.g. `qsub` or `srun`. In making this translation our aim is to arrive at a physical topology that maximises application performance by satisfying a number of constraints:

- Balance images across potentially heterogeneous nodes.
- Balance images across sockets for multi-socket nodes.
- Assign local memory in the correct Non-uniforn memory acess (NUMA) region to each image.
- Assign core affinities to each image.
- Suitable environment for each image to enable use of `libdmapp/libpgas` and the Aries network.

The traditional HPC workload managers such as Moab® /Torque and Slurm present on our XC systems make this relatively easy. Indeed the first version of the launcher was developed on a XC system using a Moab/Torque based workload manager. When we started running on Slurm-based systems the only difficulty was in verifying the equivalence between the set of options generated for Moab/Torque versus those generated for Slurm.

In order to be able to cope with site-specific needs, e.g. use of queues, node features etc. we also provided the ability for users to provide custom options which are passed directly to the underlying workload manager. This can be used when we encounter platforms configured differently from those we have previously tested on. A couple of examples are use of queues/partitions and use of node features/constraints. Many common constraints are supported out of the box, e.g. minimum cores and memory, by translating options provided to the launcher into the native equivalent for the platform workload manager.

3.3.1 Default Core Affinity Behaviours

One problem we have repeatedly experienced as we have gradually ported and run our software across more workload managers and different hardware configurations is their differences in behaviour with regards to default core affinity binding. By this we mean the behaviour of being able to bind a particular process to a specific core/socket and thus an associated NUMA region. This is very important for an application like ours which relies upon locality and efficient network data exchange. If processes are not properly bound and allowed to "float" between cores and sockets this can lead to significant slowdowns for two reasons:

1) Local memory access may cross NUMA regions.
2) Remote memory access requires a significant additional overhead as the PGAS runtime has to determine where the remote process and its memory is currently located.

This second item is compounded by the first. Not only must the runtime determine where the process is currently running, but once it has done that the local memory access may be across NUMA regions resulting in additional latency.

We also experience the issue that different workload managers exhibit different default core affinity behaviors and that these behaviors may change depending on the physical topology being requested. Specifically, in the case of Slurm, auto-binding in an exclusive partition only works when the number of images on a physical node is a multiple of the number of cores available. So depending on the hardware on any given system we may see very different performance if we do not explicitly specify the binding behavior to use. For example, if you ask for 16 images and you have a 16 core node the processes will automatically be created with core affinities such that each process is bound to specific core. However, if you had a 12 core node then the default behavior would not assign core affinities thereby impacting performance. Therefore wherever the platform supports it we make sure to specify an explicit core affinity behavior. These are included into our platform specific translations.

3.3.2 Implementation on Mesos

On the Urika-GX platform where we use the Apache Mesos [46] resource manager it was necessary to do a lot of additional work to provide the required functionality. Mesos does not have many of the concepts that we would expect to find in a traditional HPC scheduler e.g. core affinity, instead primarily focusing on basic resource attributes such as cores and memory. Additionally it does not have any knowledge of the Aries network and so does not understand how to configure the launched processes in order to allow them to communicate across the high-speed network. Therefore in order to support it as a workload manager it was necessary to develop an intermediate layer which we named `mrun` (short for Mesos run) that handles these aspects.

`mrun` has two main responsibilities. Firstly it handles the negotiations with Mesos in order to obtain the necessary resources to satisfy the requested application topology. Secondly it handles the configuration of the environments such that applications are launched with access to the Aries network. Resource negotiation with Mesos is actually handled by using the Marathon [47] framework which provides building blocks to make it easier to interface existing applications with Mesos. In order to provide the necessary functionality we needed to reproduce a portion of ALPS [48] capabilities within `mrun`. We were able to do this by adding a dependency upon `libalpscomm` allowing us to directly call the appropriate functions. Essentially our implementation uses Mesos to launch `mrun` wrapper processes on the target physical nodes, each process uses `libalpscomm` to configure the environment appropriately and then fork our actual processes i.e. the CGE server processes.

4 Scaling Experiments

We first demonstrate the scaling capabilities for CGE on XC by focusing on a set of synthetic benchmarks. Scaling experiments were run on an internal 6-cabinet XC development system with 1024 compute nodes of mixed node type (Haswell, Broadwell and KNL). We limited our study to a set of dual-socket 36-core Broadwell nodes, ranging in frequency from 2.1-2.3GHz, with 128GB DDR4-2400 memory per node. Scaling studies were run starting at 32 nodes up to 512 nodes.

For comparisons with Urika-GX we used a 48 node system (the largest configuration available) which has dual-socket 32-core 2.3GHz Broadwell nodes with 256GB DDR-2400 memory per node. Since a GX only has a maximum of 44 usable compute nodes we limit GX runs to 32 nodes to allow for direct comparisons to the smallest XC runs.

In all cases 16 images per node are used to run CGE. This has been found to be a sweet spot for performance by not over-subscribing the injection bandwith through PCIe into the Aries network, yet still providing sufficient computational parallelism.

The software used for these experiments was a CGE 3.0UP00 code base compiled using Cray Compiler Environment (CCE) 8.5.

4.1 Basic Graph Pattern (Pattern Matching) Scaling

The primary unit of search for the SPARQL query language is the Basic Graph Pattern, or 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. Performance on the LUBM [3] benchmark focuses primarily on the BGP, which makes this a good test suite for testing the scaling efficiency for the core pattern search capability of CGE.

Figure 3 shows the scaling performance of the 14 LUBM queries using 32, 64, 128, 256, and 512 nodes on our internal Cray XC-40 system. Execution time is plotted on a logarithmic scale because execution times varies considerably across the 14 queries due to differences in query complexity. Execution time reported is the strict query time and does not include the time required for writing the results to the Lustre file system. This is common practice for this benchmark since at these scales several of the queries are expected to produce hundreds of millions to billions of results, thus the IO time would dwarf the time to compute query results and the results would benchmark the system IO not compute performance. LUBM25K is a relatively moderate-sized benchmark at approximately 3 billion quads. Scaling remains quite good up through 256 nodes, although we do see scaling is starting to taper off between 256 and 512 nodes. For the larger LUBM dataset, LUBM200K, at approximately 24 billion quads, Figure 4 show strong scaling up to 512 nodes.

Note that the two most complex queries, queries 2 and 9, do not exhibit as strong scaling as the other queries, tapering

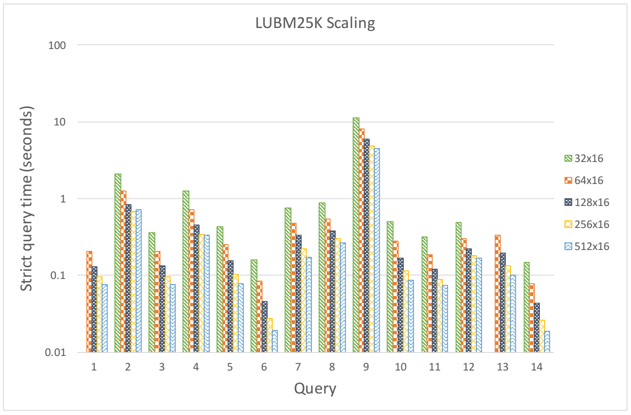

Fig. 3. LUBM25K Scaling

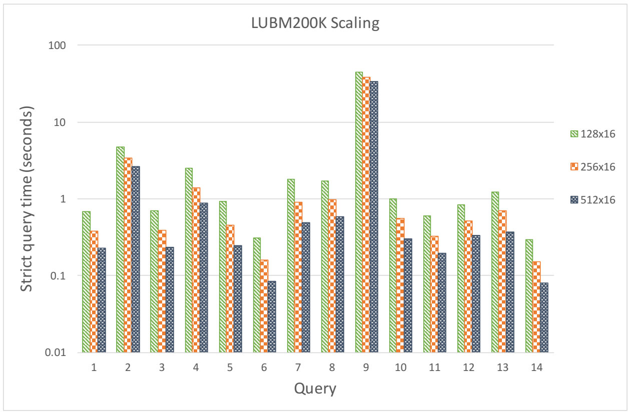

Fig. 4. LUBM200K Scaling

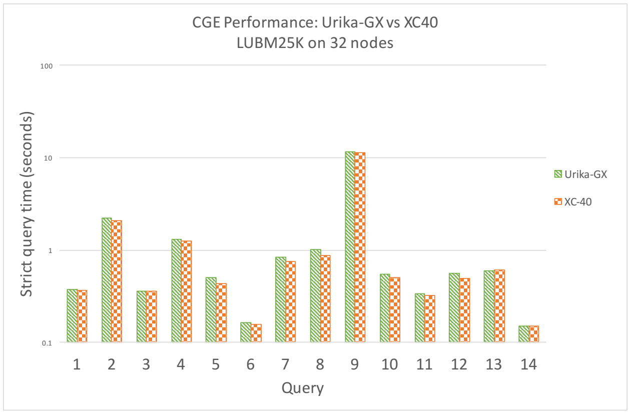

Fig. 5. LUBM25K Performance Comparison XC vs. Urika-GX

off in performance as we go from 256 to 512 nodes. These queries in particular search for triangles within the graph and as discussed later in section 5 appear to encounter a scaling limitation in the merge phase of the BGP evaluation.

Figure 5 shows a performance comparison of Urika-GX to XC for the LUBM25K benchmark run across the same number of nodes. As expected we see that Urika-GX is only slightly slower on the majority of the queries. This is primarily due to the additional network overhead. As shown earlier in Figure **??** the Fast Memory Access (FMA) throughput upon which CGE relies is essentially identical

TABLE 1
SNAP Datasets used for Study

Dataset	Description
cit-patents	Citation network among US Patents
soc-LiveJournal1	LiveJournal online social network
com-Friendster	Friendster online social network

between platforms. Therefore the only real difference is the extra latency in the physical network which results in a minimal slowdown.

4.2 Using Graph algorithms to measure global properties

To evaluate the performance of the whole graph analysis components of CGE we use a set of graphs available from the Stanford Network Analysis Project (SNAP) [4] listed in Table 1. Properties for each of these graph are shown in Table 2. These graphs are snapshots of real world datasets representing commonly encountered and analyzed graphs.

The SNAP repository provides a wide selections of different sized graphs. For this study we were most interested in the moderate to larger sized graphs where the number of vertices was greater than 1 million. The graphs provided by SNAP are simple integer edge lists. To import into CGE, we used a simple code to convert these to an N-Triples format using Spark. For each of the datasets we pre-built the database from the N-Triples generated N-Triples file. We did not include the build or load time of the database, but focused on the query time for running each of the graph algorithms.

4.2.1 Algorithms

We looked at the performance of a small set of three commonly used graph analysis algorithms: degree distribution, PageRank [1], and triangle counting. Implementations for these algorithms are available in most graph mining systems.

PageRank is a widely used graph analysis algorithm which was originally developed as a way to measure the importance of website pages. The use of PageRank has expanded to other graph analysis applications to measure the importance of vertices in a graph. In addition to a configurable damping factor (typically set at 0.85), there exist multiple variations and simplifications of the PageRank algorithm.

The current Built-In Graph Function (BGF) [49] implementation of PageRank includes the rank adjustment for leaf-nodes, or vertices with zero out-degree, while the "Python+SPARQL" implementation did not. To make the comparisons more apples-to-apples, we are including the bi-directional edges in the BGP, such that the graphs are essentially undirected with no zero out-degree vertices. The SPARQL query for this can be seen in Listing 1.

Degree distribution provides the number of vertices with a specific degree count. This can be achieved purely through standard SPARQL features as shown in Listing 2 using the `GROUP BY` and aggregation features of the language. This provides a quick measure of the connectivity and distribution of the graph. This is often used as a basis for further analysis, for example are there a small number of vertices with very high degree thus implying hot spots in the graph. Additionally it can be used in ranking algorithms to compare a given vertex against the global distribution.

Triangle Counting also plays an important function in graph analysis. Triangles are the most basic non-trivial subgraphs in graphs. Many social networks, for example, contain lots of triangles, where friends of friends tend to become friends themselves. This pattern is also observed in other types of networks as well such as biological and online networks (Web graphs).

For Triangle Counting, the counts provided by the SNAP repository consider the network as undirected. Our current BGF implementation for triangle counting currently targets only directed graphs, so to generate an undirected graph we also include the bi-directional edges. We also include the added restriction that our Triangle Counting queries only counts distinct triangles composed of unique sets of three vertices. This allows us to directly compare the triangle counts with the counts provided in the SNAP repository. Listing 3 shows the SPARQL query for Triangle Counting which uses the CGE BGF. You can see the bi-directional edges included in the BGP portion within the `CONSTRUCT` and the additional `FILTER` operation required for counting only distinct triangles.

The "Python+SPARQL" implementation of the query is similar, but here we first need to save the results from the same BGP into a named graph as part of an optimization step between iterations.

```
PREFIX cray:   <http://cray.com/>

SELECT ?vertex ?rank
WHERE{
  CONSTRUCT {
    ?vertex1 ?edge ?vertex2 .
  }
  WHERE {
    { ?vertex1 ?edge ?vertex2 . }
    UNION
    { ?vertex2 ?edge ?vertex1 . }
  }
  INVOKE cray:graphAlgorithm.pagerank(0.0001,0.85)
  PRODUCING ?vertex ?rank
}
ORDER BY DESC(?rank)
```

Listing 1. PageRank using SPARQL w/ BGF

```
  SELECT ?degree (COUNT(?degree) AS ?count)
  WHERE
  {
    {
      SELECT (COUNT(?vertex) AS ?degree)
      WHERE
      {
        { ?vertex <urn:connectedTo> ?outgoing . }
        UNION
        { ?incoming <urn:connectedTo> ?vertex . }
      }
      GROUP BY ?vertex
    }
  }
  GROUP BY ?degree
  ORDER BY ?degree
```

Listing 2. Degree Distribution using SPARQL

```
  PREFIX cray:   <http://cray.com/>

  SELECT ?total_num_triangles
  WHERE{
    CONSTRUCT {
      ?vertex1 ?edge ?vertex2
    }
    WHERE {
      { ?vertex1 ?edge ?vertex2 }
      UNION
      { ?vertex2 ?edge ?vertex1 }
      FILTER(?vertex1 < ?vertex2)
    }
    INVOKE   cray:graphAlgorithm.triangle_counting(2)
    PRODUCING ?total_num_triangles
```

TABLE 2
Graph Metrics for SNAP Datasets

Dataset	Vertices	Edges	Triangles	Unique Degrees
cit-patents	3,774,768	16,518,948	7,515,023	370
soc-LiveJournal1	4,847,571	68,993,773	285,730,264	2,045
com-Friendster	65,608,366	1,806,067,135	4,173,724,142	3,148

```
}
```

Listing 3. Triangle Counting Query using SPARQL w/ BGF

4.2.2 *Graph Algorithm Results*

For degree distribution, which is expressible in pure SPARQL, we do not provide an optimized implementation since it is a single query. Therefore there is also no need to look at iterative approaches for this algorithm because there would only be a single iteration. Performance for this algorithm can be seen in Figure 6 where we see that this query exhibits limited scaling.

The problem with degree distribution is that the scalability is fundamentally limited by the characteristics of the dataset. The query is in essence a two stage reduction. In the first stage we reduce by vertex, and per Table 2 we have millions of vertices so there is plenty of parallelism available at this stage. However, in the second stage we reduce by degree, of which there are typically only a few thousand unique degrees. As can be seen in Table 2 even the larget dataset, Friendster, only has 3,148 unique degrees. Thus there is a limit to the amount of parallelism we can usefully apply in this second stage. As we see in Figure 6 once we use a sufficient number of nodes such that there are more images available than unique degrees performance actually starts to worsen. This is because each group key, in this case the unique degree value, is hashed to a specific image and data for each group is aggregated at that image. When there are more images available than unique group keys we inherently create load imbalance. Some proportion of images will have zero work to do while others may have to compute many groups because as with any hash function it is imperfect. Despite this the analysis can still be run in interactive timeframes. It may be possible to achieve further improvements by further optimizing the implementation of the `GROUP BY` operator though as noted the nature of the dataset may place a limit on scaling.

For PageRank and Triangle Counting we provide a performance comparison between two different approaches both available within CGE. The first approach, which we refer to as "Python+SPARQL", uses the approach presented in [50] which maps an iterative algorithm to a combination of SPARQL queries whose execution sequence is managed using a procedural language such as Python or Javascript. Intermediate results produced during the computation are saved as temporary named graphs within the running database. Here we were able to leverage the implementations for PageRank and Triangle Counting discussed in [51]. The second approach is to use the optimized built-in graph function (BGFs) for PageRank and Triangle Counting provided by CGE [49]. These leverage an extension to the

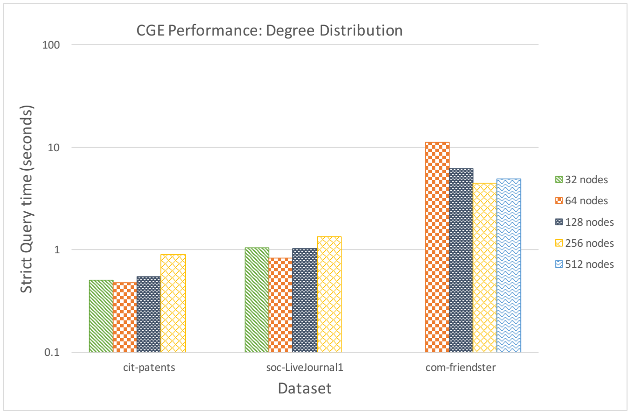

Fig. 6. CGE Performance: Degree Distribution

language that allows us to call native Coarray C++ implementations of the algorithms from within a SPARQL query. This allows for a standard query to be used to identify the portion of the graph to be analyzed before the native algorithm is invoked.

In Figures 7 and 8 you can see the scaling results for PageRank using the two approaches. We can see that for the smaller datasets the BGF approach is 2 orders of magnitude faster and 1 order of magnitude faster for the largest dataset. We would also note that the BGF approach exhibits strong scaling whereas the "Python+SPARQL" approach yields very little benefit as we scale up.

Figures 9 and 10 show the scaling results for triangle counting using the two approaches. Here we can say that while the BGF approach is an order of magnitude faster we achieve limited scaling on this algorithm. We are investigating why triangle counting does not scale up currently and have a couple of potential culprits. One potential issue is that we consider candidate triangles in batches and the choice of batch size may be sub-optimal for larger node counts. Another issue is that the characteristics of the graph may be causing hot spotting and load imbalance whereby some processes have far more work to do than others. Despite the lack of scaling we are again able to complete these queries in interactive time.

4.3 Comparing CGE to Spark GraphX

In [52] they presented a comprehensive comparison between several graph analysis ecosystems. Their conclusions were that the stand-alone systems like NetworkX, Neo4j and Apache Jena provided good performance for small scale graphs, but only scalable solutions, such as Apache Spark, GraphX and Urika®-GD were able to provide interactive level query response times for larger graphs, with

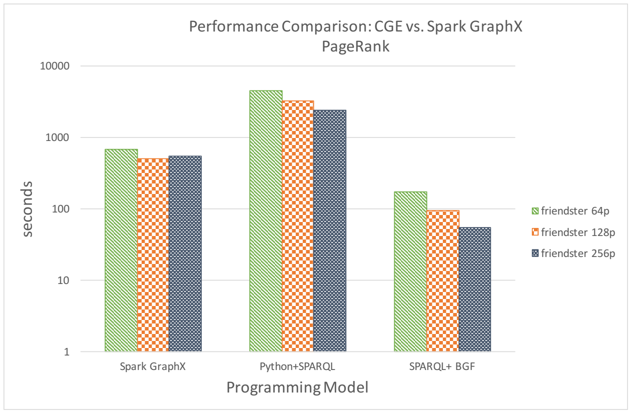

Fig. 7. CGE Performance: PageRank using BGF

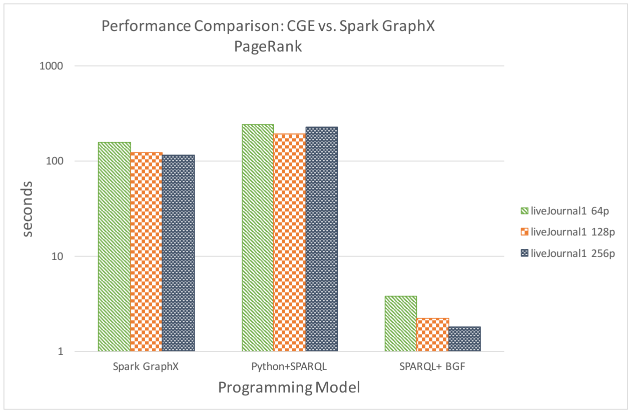

Fig. 8. CGE Performance: PageRank using Python+SPARQL

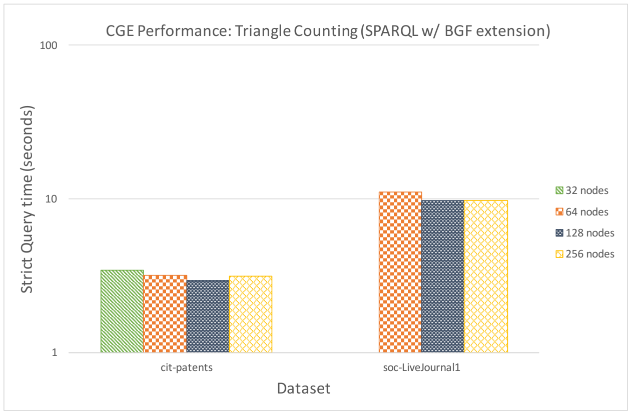

Fig. 9. CGE Performance: Triangle Counting using BGP

the Cray Urika®-GD system providing the most scalable performance. The authors of [52] also graciously provided us with their Spark GraphX implementations for the LUBM queries.

In mid 2016 Databricks introduced GraphFrames [53], a graph processing library based on Spark DataFrames which over time may replace the Spark GraphX library. With better support provided for Spark DataFrames in Spark 2.0, we did some initial investigation into rewriting the Spark GraphX LUBM queries, based on Spark GraphRDD triplets, to use GraphFrames, based on Spark DataFrames instead. Graph-

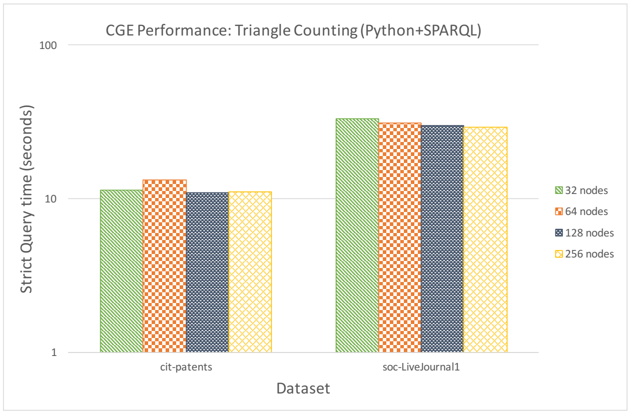

Fig. 10. CGE Performance: Triangle Counting using Python+SPARQL

Frames uses a query syntax based on the Cypher query language used by Neo4j. What we found at the time was that our GraphFrames implementations of the LUBM queries were able to generate correct results for simple pattern matching queries such as Query 1, but returned incorrect results for the more complex trianglular queries such as Query 2 and Query 9. We do plan to continue to monitor the GraphFrames project as the implementation continues to mature, therefore for this performance comparison we only use the Spark GraphX-based queries.

One limitation which the authors pointed out for the Cray Urika-GD system was the size of the shared memory, noting that performance could hit a wall once the dataset exceeded the size of the memory. With CGE on Cray XC, we are now able to scale out to much larger node configurations and memory sizes. As an extension to their earlier work, we show the performance of LUBM25K using Spark GraphX on XC and compare this performance to that of CGE.

We run Spark 2.1.0 using the Beta version of Cray's forthcoming analytics on XC package. We deploy Spark as a set of containers running in the Shifter containerizer [54] on compute nodes of our Cray XC system:

- One node runs a container with a Spark master image. The master schedules work across the executor (worker) nodes.
- One node runs an interactive image that users may use to interact with the Spark cluster via Java, Scala, Python, R or SQL.
- The remaining allocated nodes run Spark executors (workers) inside Shifter containers. These executors execute tasks on partitions of the user's data.

We configure our Shifter containers to utilize Shifter's per-node cache feature to provide local temporary storage. The per-node cache provides an XFS loopback mounted filesystem to each container, backed by a file on Lustre [55]. This eliminates one of the primary bottlenecks seen with running frameworks like Spark on compute-dense HPC systems with little or no local storage—namely, the poor performance of shared storage when used to emulate the per-node local storage which these frameworks depend on for spilling and shuffling. This poor performance is due to two primary causes: difficulties with OS caching, and bottlenecks on the shared storage metadata due to many

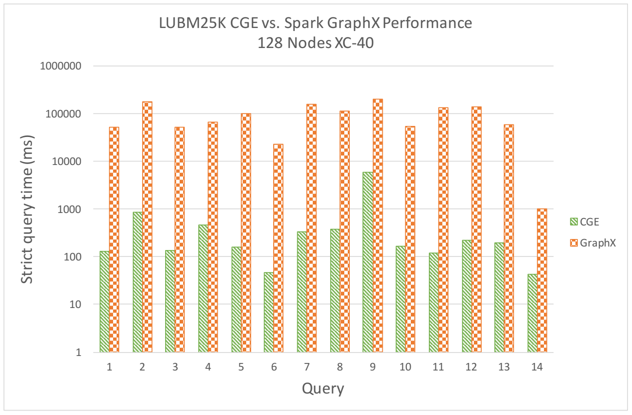

Fig. 11. CGE vs Spark Performance: LUBM25K

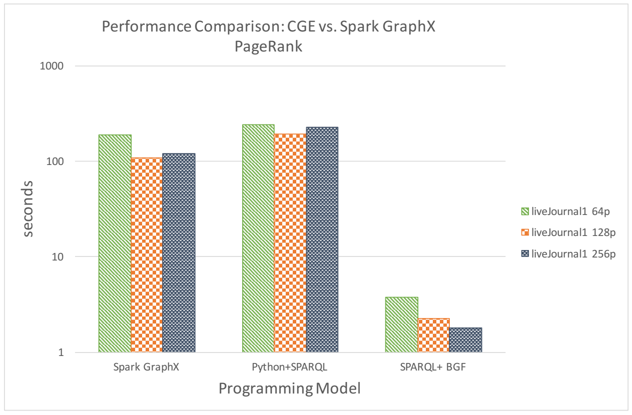

Fig. 12. CGE vs Spark Performance: PageRank - LiveJournal Dataset

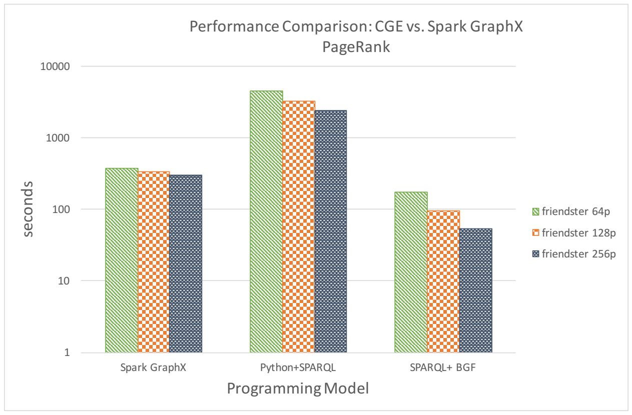

Fig. 13. CGE vs Spark Performance: PageRank - Frienster Dataset

small file accesses on multiple worker nodes. By providing a separate, effectively-local filesystem to each node, Shifter's per-node cache eliminates both of these issues.

Typically Spark applications maximise performance by co-locating the computation with the data. Since the data is stored on the parallel Lustre filesystem it will not be co-located with the computation initially. To mitigate this our spark codes first load in the data from the parallel filesystem and then explicitly perform a Spark `persist()` operation, this instructs Spark to cache the data in-memory on the compute nodes. Thus the actual computation operates fully in memory on co-located data. Where shuffles are necessary the temporary data is written to the aforementioned loopback filesystem. Since these files are typically small and short lived they usually remain in OS disk cache in-memory and they are not flushed to physical storage.

Using the LUBM [3] benchmark suite, which exercises the graph pattern matching capabilities of the SPARQL query engine, is certainly favorable to CGE, since this is what is was designed to do well. Figure 11 highlights the significant performance advantage observed when comparing Spark GraphX performance to CGE performance with the moderately sized LUBM25K dataset. Graph pattern matching can be both latency sensitive and communication intensive. Although both Spark and CGE communicate over the high-performance Aries network on XC, Spark communication uses TCP/IP, while CGE uses the lower-level Distributed Memory Application (DMAPP) [56] communication layer.

For exploratory graph analysis, Spark GraphX, although still an order of magnitude slower than CGE, is much more competitive. Here we compare the performance of running PageRank using Spark GraphX, and compare this to the two different SPARQL implementations, the "Python+SPARQL" PageRank implementation and the CGE BGF PageRank implementation. As one would expect, the customized native Coarray C++ implementation of the PageRank BGF provided CGE with outperforms both the "Python+SPARQL" approach and the Spark GraphX implementation. For purposes of comparison we use the faster static iterations version of PageRank provided in Spark. The number of iterations used is the number of iterations it takes the CGE BGF implementation, which does convergence checking, to converge to a solution. Thus creating an apples-to-apples comparison of the algorithm running the same number of iterations.

However, as can be seen in Figures 12 and 13 the difference in performance varies considerably depending on the dataset. For the smaller LiveJournal dataset shown in Figure 12 we see that the BGF implementation is clearly orders of magnitude better achieving times approximately 100x that of the Spark GraphX implementation running in 1.79 seconds versus 115.5 seconds (taking the best times achieved). However, in the larger and more complex Friendster graph shown in Figure 13 the BGF implementation is clearly better the performance ratio is around 10x that of Spark GraphX running in 53.84 seconds versus 500 seconds.

CGE and Spark are both supported components of the forthcoming Cray analytics stack for XC and we recently added a new feature to the CGE distribution, the CGE Spark API, which helps make moving data between CGE and Spark more seamless. This feature allows Spark programmers to convert CGE results files in tab-separated-values format (".tsv" files) into Spark DataFrames. For more details, see the CGE Users Guide [57].

5 UNIPROT RESULTS

The UniProt [5] database is the authoritative collection of functional information on proteins, and includes annotations, interrelationships and in some cases the amino acid

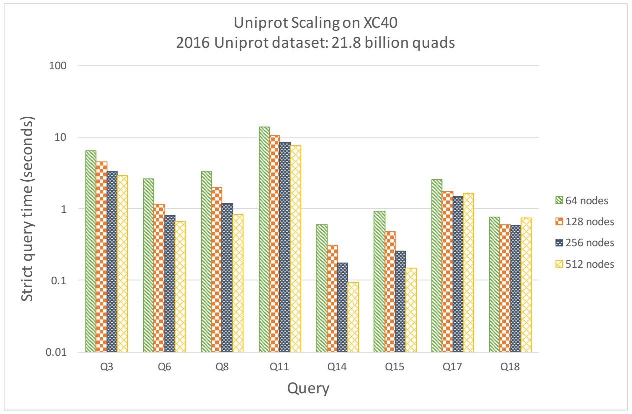

Fig. 14. CGE Performance: Uniprot dataset queries

sequences of the proteins themselves. Proteins are the building blocks of all life on earth, and the Uniprot database is crucial to Life Science researchers hoping to understand them. Uniprot concentrates on Human proteins, though other widely studied organisms such as rice, mouse and zebrafish are also well represented.

The UniProt Consortium is a collaboration between the European Bioinformatics Institute (EBI), the Swiss Institute of Bioinformatics (SIB) and the Protein Information Resource (PIR). It has been a pioneer in Semantic Web technology, and Uniprot has been distributed in RDF format since 2008. New releases are distributed every four weeks. The interactions between proteins are complex and widely linked, so a graph representation is very useful.

Uniprot is continually growing as more scientific data is added and the very latest release contains 27,291,595,271 triples (27 Billion). The database we used for this investigation is from mid-2016 and contains 22 Billion triples. In the form of an N-Triples (.nt) file on disk it is roughly 3.8 Terabytes. Including sections for different organisms, there are 17 named graphs.

For our study we looked at the scaling performance of eight queries applied to the Uniprot dataset as shown in Table 3. Six of the queries are example queries provided from the Uniprot SPARQL endpoint [58]. We also included two queries provided by Eric Neumann, which were provided as sample hands-on examples at the Semantic Web Interest Group, June 2009 Workshop [59]. These queries were specifically selected to demonstrate more complex BGPs and query structure than the synthetic benchmarks.

Although we continue to scale up to 512 nodes on six of the eight queries, scaling tapers off significantly between 256 nodes and 512 nodes. Focusing in on Query 11 (shown in Listing 4), which takes the longest amount of time, we broke down the amount of time spent in each phase of the BGP (`SCAN`, `JOIN`, `MERGE`) portion of the Query. We used this data to identify which phase is a current limitation. Table 4 shows that the `SCAN` and `JOIN` phases are scaling well, but the `MERGE` phase is clearly limiting scaling. The `SCAN` phase shows strong scaling all the way to 512 nodes, and the `JOIN` phase exhibits good scaling up to 512 nodes though it is clearly tapering off. However, the `MERGE` performance is essentially identical across all node counts showing that it is exhibiting no scaling.

In the `SCAN` phase, we find candidate solutions for each query quad pattern, a pattern that matches edges in the graph, in the BGP by searching through the local portion of the database residing on that image. Each `SCAN` is embarrassingly parallel and communication between images is only needed to balance the solutions across images after each image has performed their local scan operation. As already described in Section 3.2, this balancing is implemented such as to minimize network data exchange. The

TABLE 3
Uniprot Database queries used for study

Query	Origin	Description
Q3	Example 3 http://sparql.uniprot.org	Select all E-Coli K12 Uniprot (including strains) entries and their amino acid sequence
Q6	Example 6 http://sparql.uniprot.org	Select all cross-references to external databases of the category ’3D structure databases’ of UniProt entries that are classified with the keyword ’Acetoin biosynthesis (KW-0005)’
Q8	Example 8 http://sparql.uniprot.org	Select the preferred gene name and disease annotation of all human UniProt entries that are known to be involved in a disease
Q11	Example 11 http://sparql.uniprot.org	Select all UniProt entries with annotated transmembrane regions and the regions’ begin and end coordinates on the canonical sequence
Q14	Query 9 Eric Neumann 2009 Workshop	Finding Proteins with Reactome associations
Q15	Query 15 Eric Neumann 2009 Workshop	Finding Proteins with both Reactome and Citation associations to each other
Q17	Example 17 http://sparql.uniprot.org	Select the average number of cross-references to the PDB database of UniProt entries that have at least one cross-reference to the PDB database
Q18	Example 18 http://sparql.uniprot.org	Select the number of UniProt entries for each of the EC (Enzyme Commission) second level categories

```
#Select all UniProt entries with annotated transmembrane regions
# and the regions' begin and end coordinates on the canonical sequence

PREFIX up:<http://purl.uniprot.org/core/>
PREFIX faldo:<http://biohackathon.org/resource/faldo#>

SELECT ?protein ?begin ?end
WHERE
{
    ?protein a up:Protein .
    ?protein up:annotation ?annotation .
    ?annotation a up:Transmembrane_Annotation .
    ?annotation up:range ?range .
    ?range faldo:begin/faldo:position ?begin .
    ?range faldo:end/faldo:position ?end
}
```

Listing 4. Uniprot Query 11: Annotated transmembrane regions

TABLE 4
BGP time breakdown for Uniprot Query 11 (seconds)

Phase (Ops)	64 nodes	128 nodes	256 nodes	512 nodes
SCAN (8)	5.74	3.12	1.76	0.98
JOIN (5)	1.73	0.98	0.65	0.53
MERGE (6)	5.86	6.17	5.94	6.10
Total	14.01	10.60	8.52	7.70

result of the `SCAN` phase is a list of solution sets, known as Intermediate Result Arrays (IRAs), where each IRA represents the matches for a specific scan.

In the `JOIN` phase, we attempt to reduce the size of the candidate solutions returned by the `SCAN` phase by comparing the variable bindings across multiple IRAs. `JOIN` is just a simple unary association of the `SCAN` output on a single variable. For each unbound variable (UBV) that appears in multiple IRAs, we determine which internal identifiers, known as HURIs, are valid for that UBV. To be valid, a HURI must appear in at least one solution of all solution sets that contain that UBV. This allows eliminating solutions which cannot possibly be merged in the subsequent `MERGE` phase thus reducing the solution space.

The final `MERGE` phase is essentially a merge join of the IRAs. It first uses a complex heuristic to try to determine an optimal schedule, i.e. an ordering in which to perform the sequence of merges. Communication is all-to-all, in that during the merge every image must send its data to all the other images. We have optimized this at a node level so images within the same physical node share any data which was read from a remote image. The algorithm then becomes a nested loop of the form shown in Listing 5.

```
for(int64_t grp = 0; grp < num_groups; grp++) {
  for(int64_t grp_img = 0; grp_img < num_images_in_grp; grp_img++)   {
    // Do merge work
  }
}
```

Listing 5. Psuedo-code for MERGE phase

The outer most loop is a loop over the number of groups. The number of outer loop iterations increases as we add more nodes and communication/global synchronization is required between each iteration. We do pre-fetching to overlap the communication with the on-node computation. We also plan to look at taking advantage of the forthcoming availablity of synchronization within sub-teams of images. This would reduce the amount of global synchronization. Our loop nest would now be a triple loop nest, with the outer loop a loop over teams, the middle loop over groups, and the inner most loop still over images within the same physical node.

Within the local computation, the memory access patterns are very much data driven, and the number of potential matches or solutions is unknown. Since this is a merge join the amount of work depends upon both how many intermediate solutions must be sorted but also upon the selectivity of the merge join. If a merge join has low selectivity it produces many merges which requires scanning large portions of the data to find all compatible solutions to be merged. This can lead to an explosion of the intermediate solution space as we proceed through the merges. As already noted, we use a heuristic to schedule the order of the merges. This tries to prefer high selectivity merges i.e. those that will reduce the size of the intermediate solution space. However, it is difficult to estimate in advance the selectivity of a given merge join so we may not always pick an ideal schedule. In the worst pathological cases where a merge has no common variables we are forced to compute a cross product which squares the solution space. If multiple cross products are encountered the solution space may grow exponentially during the course of the `MERGE` phase.

In the cases of merge joins that substantially expand the solution space we suspect that we are defeating the processor cache leading to cache thrashing. Each image is sequentially reading through the currently node-local portions of the data, and may repeat this multiple times. So where we have large solution spaces much of the memory read will have been evicted from cache by the time it is needed again. Therefore one area of additional performance optimization is to look at the current cache performance when performing the local merge phases. This will involve the use of low level profiling via CrayPAT and based on these findings we will look at ways to improve memory reuse or to avoid thrashing the cache.

6 Conclusion

In conclusion we have presented CGE and described the challenges and successes we have had porting to run across multiple hardware platforms. We have been able to demonstrate the ability for an application to strongly scale up from our Urika-GX platform onto our XC platform. Thus providing a clear upgrade path of customers as their datasets and workload increase in size and complexity.

Through experiments we have shown that CGE exhibits strong scaling across a variety of graph analytics workloads. Compared to leading open source competitors we are able to achieve substantially better performance on those workloads and scale performance to massive graphs. We exhibit 10 to 100 times better performance than those solutions depending on the workload and dataset. This enables interactive analyses that were previously not possible at scale.

Our results demonstrate that CGE is capable of strong performance across the two kinds of graph analysis workloads where traditionally systems have only targeted one. This shows that is it possible to engineer a system that is able to support both workloads without compromising on performance. This is a significant improvement over previous systems which either limited users to one workload, or forced them to compromise on performance.

Finally, we have identified areas where we still have further scaling work to do. Where time allowed we have provided preliminary root cause analysis for these. Additional work is still required to confirm these analyses and address these limitations. However, as has been discussed some of these limitations are data and workload dependence, as such scope for improvement may vary.

7 Future Work

As explained in Section 3.2, we aim to maximize performance of our application by maximizing locality when possible. Currently we are somewhat limited by the Coarray C++ runtime in that it only provides for global synchronization across all images. There are many places where we would be better served by performing a more constrained synchronization, e.g. within the images of a given physical node or within a small subset of nodes. The developers of Coarray C++ within Cray are currently working on implementing a teams feature which will allow images to be subdivided into teams based upon arbitrary criteria e.g. physical node ID. This should allow us to further optimize our application by reducing the number of global synchronizations required in favor of more local synchronizations.

We will also be looking to investigate how to resolve some of the current limitations on scaling identified in this paper. In particular we will focus on the merge code that we know to be a problem per Section 5. Since that operation is used in almost every query any performance improvements there will yield benefits across many workloads.

Acknowledgments

The authors would like to thank all the members of the CGE team who have contributed to the development of this software and in particular those who have helped in the preparation and review of this paper. We would also like to thank all the customers who have used CGE and provided us with valuable feedback that has allowed us to continue to iterate and improve the software.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[2] 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.

[3] 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.

[4] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[5] U. Consortium, "Uniprot: the universal protein knowledgebase," *Nucleic Acids Research*, vol. 45, no. D1, p. D158, 2017. [Online]. Available: http://dx.doi.org/10.1093/nar/gkw1099

[6] P. Uetz, L. Giot, G. Cagney, T. A. Mansfield, R. S. Judson, J. R. Knight, D. Lockshon, V. Narayan, M. Srinivasan, P. Pochart *et al.*, "A comprehensive analysis of protein–protein interactions in saccharomyces cerevisiae," *Nature*, vol. 403, no. 6770, pp. 623–627, 2000.

[7] W. Fan, "Graph pattern matching revised for social network analysis," in *Proceedings of the 15th International Conference on Database Theory*. ACM, 2012, pp. 8–21.

[8] V. Chandola, S. R. Sukumar, and J. C. Schryver, "Knowledge discovery from massive healthcare claims data," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1312–1320.

[9] A.-L. Barabâsi, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, and T. Vicsek, "Evolution of the social network of scientific collaborations," *Physica A: Statistical mechanics and its applications*, vol. 311, no. 3, pp. 590–614, 2002.

[10] G. A. Pagani and M. Aiello, "The power grid as a complex network: a survey," *Physica A: Statistical Mechanics and its Applications*, vol. 392, no. 11, pp. 2688–2700, 2013.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[12] C. C. Aggarwal, J. L. Wolf, K.-L. Wu, and P. S. Yu, "Horting hatches an egg: A new graph-theoretic approach to collaborative filtering," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 201–212.

[13] C. C. Aggarwal, H. Wang *et al.*, *Managing and mining graph data*. Springer, 2010, vol. 40.

[14] S. Even, *Graph algorithms*. Cambridge University Press, 2011.

[15] T. Washio and H. Motoda, "State of the art of graph-based data mining," *Acm Sigkdd Explorations Newsletter*, vol. 5, no. 1, pp. 59–68, 2003.

[16] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM computing surveys (CSUR)*, vol. 38, no. 1, p. 2, 2006.

[17] L. B. Holder, D. J. Cook, S. Djoko *et al.*, "Substucture discovery in the subdue system." in *KDD workshop*, 1994, pp. 169–180.

[18] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 721–724.

[19] L. Akoglu, M. McGlohon, and C. Faloutsos, "Oddball: Spotting anomalies in weighted graphs," *Advances in Knowledge Discovery and Data Mining*, pp. 410–421, 2010.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[21] D. A. Schult and P. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, vol. 2008, 2008, pp. 11–16.

[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, "Graphlab: A distributed framework for machine learning in the cloud," *arXiv preprint arXiv:1107.0922*, 2011.

[24] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner, *Neo4j in action*. Manning, 2015.

[25] "Titan distributed graph database," Jun. 2017, accessed 4/4/2017. [Online]. Available: http://thinkaurelius.github.io/titan/

[26] B. Shao, H. Wang, and Y. Li, "The trinity graph engine," *Microsoft Research*, p. 54, 2012.

[27] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel, "An experimental comparison of rdf data management approaches in a sparql benchmark scenario," in *International Semantic Web Conference*. Springer, 2008, pp. 82–97.

[28] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *International semantic web conference*. Springer, 2002, pp. 54–68.

[29] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The ics-forth rdfsuite: Managing voluminous rdf description bases," in *Proceedings of the Second International Conference on Semantic Web-Volume 40*. CEUR-WS. org, 2001, pp. 1–13.

[30] Z. Liu, A. Le Calvé, F. Cretton, and N. Glassey, "Using semantic web technologies in heterogeneous distributed database system: A case study for managing energy data on mobile devices," *International Journal of New Computer Architectures and their Applications (IJNCAA)*, vol. 4, no. 2, pp. 56–69, 2014.

[31] X. Zhang, L. Chen, Y. Tong, and M. Wang, "Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud," in *Data engineering (ICDE), 2013 ieee 29th international conference on*. IEEE, 2013, pp. 565–576.

[32] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "Triad: a distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 289–300.

[33] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered rdf store," in *5th International Workshop*

on Scalable Semantic Web Knowledge Base Systems (SSWS2009), 2009, pp. 94–109.

[34] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "Yars2: A federated repository for querying graph structured data from the web," *The Semantic Web*, pp. 211–224, 2007.

[35] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.", 2015.

[36] A. Seaborne and S. Harris, "SPARQL 1.1 query language," W3C, W3C Recommendation, Mar. 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[37] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 195–204.

[38] M. San Martın, C. Gutierrez, and P. T. Wood, "Snql: A social networks query and transformation language," *cities*, vol. 5, p. r5, 2011.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[40] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[42] Y. Sun and J. Han, "Mining heterogeneous information networks: principles and methodologies," *Synthesis Lectures on Data Mining and Knowledge Discovery*, vol. 3, no. 2, pp. 1–159, 2012.

[43] P. Erdös and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 17-61, p. 43, 1960.

[44] J. Celko, *Joe Celko's SQL for smarties: advanced SQL programming*. Elsevier, 2010.

[45] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[46] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[47] "Marathon: A container orchestration platform for mesos and dc/os," Jun. 2013, accessed 3/20/2017. [Online]. Available: https://mesosphere.github.io/marathon/

[48] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, "The application level placement scheduler," *Cray User Group*, pp. 1–7, 2006.

[49] D. Mizell, K. J. Maschhoff, and S. P. Reinhardt, "Extending sparql with graph functions," *2014 IEEE International Conference on Big Data*.

[50] R. Techentin, B. Gilbert, A. Lugowski, K. Deweese, J. Gilbert, E. Dull, M. Hinchey, and S. Reinhardt, "Implementing Iterative Algorithms with SPARQL," *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference*, pp. 216–223.

[51] S. Lee, S. R. Sukumar, and S.-H. Lim, "Graph Mining Meets the Semantic Web," in *Proceedings of ICDE Workshop on Data Engineering meets the Semantic Web*, 2015.

[52] S. Hong, S. Lee, S.-H. Lim, S. R. Sukumar, and R. R. Vatsavai, "Evaluation of pattern matching workloads in graph analysis systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 263–266.

[53] "Spark graphframes 0.4.0 user guide," Tech. Rep. [Online]. Available: http://graphframes.github.io/user-guide.html

[54] D. Jacobsen and R. Canon, "Shifter: Containers for hpc," in *Cray Users Group Conference (CUG16)*, 2016.

[55] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Performance evaluation of apache spark on cray xc systems," May 2016.

[56] M. ten Bruggencate and D. Roweth, "Dmapp-an api for one-sided program models on baker systems," 2010.

[57] *Cray Graph Engine User Guide (3.0UP00) S-3014*, March 2017. [Online]. Available: http://docs.cray.com/PDF/Cray_Graph_Engine_User_Guide_30UP00_S-3014.pdf

[58] "Uniprot sparql endpoint," Tech. Rep. [Online]. Available: http://sparql.uniprot.org

[59] E. Neumann, "Sparql: Some hands-on examples using twinkle and uniprot," Tech. Rep. [Online]. Available: https://sites.google.com/site/houstonsemweb/program/uniprot