THE SUPERCOMPUTER COMPANY

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

April 27th 2015
Kristyn Maschhoff
James Maltby
Robert Vesse

Outline of Talk

- Introduction to Urika-GD
- Porting the Query Engine
- Performance - Urika-GD vs. XC40
- Porting the User Interface
- Bioinformatics Use Case
- Conclusion

Legal Disclaimer

Urika-GD Graph Discovery Appliance

Urika-GD

Standalone Semantic Database Appliance	• **Integrated** semantic database software, server, and storage • **Designed and Optimized** for graph analytics • **Standards Based** (W3C/Linux + RDF/SPARQL)
Unique Hardware Platform	• **Threadstorm** Multithreaded Processors • Cray **SeaStar2** interconnect • **Uniform Shared Memory** programming model
Enterprise Software	• **SPARQL 1.1** graph query engine • Browser-based Database front end
Cray SPARQL Extensions	• **B**uilt-in **G**raph **F**unctions (**BGFs**)

Purpose Built for Graph Data Discovery

Motivation

- **Cray provides two strong product offerings in Analytics with Urika-GD and Urika-XA**
  - Extensive interest in Analytics on HPC
- **Extending the Graph Analytics capabilities developed for the Urika-GD to the XC30/40 line**
  - Cray Graph Engine (CGE)
  - Enables mixed workloads
    - Traditional HPC (simulation)
    - Graph Discovery (Urika-GD)
  - Creation of complex analytics workflows
    - Multi-step and mixed analytic workflows
    - Spark + CGE
  - Shared high performance network and storage

Porting the Query Engine - Early Investigations

- **Data in an RDF database is unstructured**
  - Communication of information across the dataset can be highly irregular, may approach all-to-all for tightly connected graphs
  - Maintaining optimal network performance for short remote references, both PUTs and GETs is essential
  - Urika-GD does this very well
    - ~100 Mrefs/s per node for single word loads and stores
    - But all references are remote!

- **Mapping to XC architecture**
  - Global address space for one-sided communication
  - Leverage the low-level DMAPP library communication layer
    - Non-blocking implicit GETs and PUTs (~50 Mrefs/s per node, single word)
  - Utilize synchronization features and atomic operations available with Aries

Selection of Coarray C++ Programming Model

- **A more standard HPC programming methodology**
  - Alternative was to develop an in-house programming environment to emulate shared memory model of XMT on XC to provide a single code base for both platforms
- **C++ template library that runs on top of Cray's Partitioned Global Address Space (PGAS) library**
  - Provides the performance advantages of the low-level DMAPP communication
  - Provides easy access to Aries synchronization features and atomic operations
  - Urika-GD codebase is currently C++

Coarray C++ Programming Model (continued)

- **Coarray provides an easy model for taking advantage of locality when available**
  - Internal intermediate data structures
- **Flexibility to add application specific functionality by customizing Coarray C++ template files**
- **Carrying forward the Basic Graph Function (BGFs) extensions**
  - Custom graph algorithms written in Coarray C++
  - Coarray C++ can also be used with Cray MPI to incorporate third party graph libraries
    - Parallel Boost Graph Library
    - Knowledge Discovery Toolkit (using Combinatorial BLAS)

Extending Coarray C++ template files

- **Started with the coarray_cpp.h template file provided with the Cray compiler**
- **Examples of customizations added for CGE**
  - mget member function to simplify syntax for multi-word gets
  - New member functions to allow vanilla loads and stores for coatomics
    - Allow these to be more lightweight in regions where we know these are not being updated or where there are no conflicts (initialization loop)
  - Threading specific member functions
    - e.g. get_switch() member function that issues a non-blocking GET followed by a thread context switch

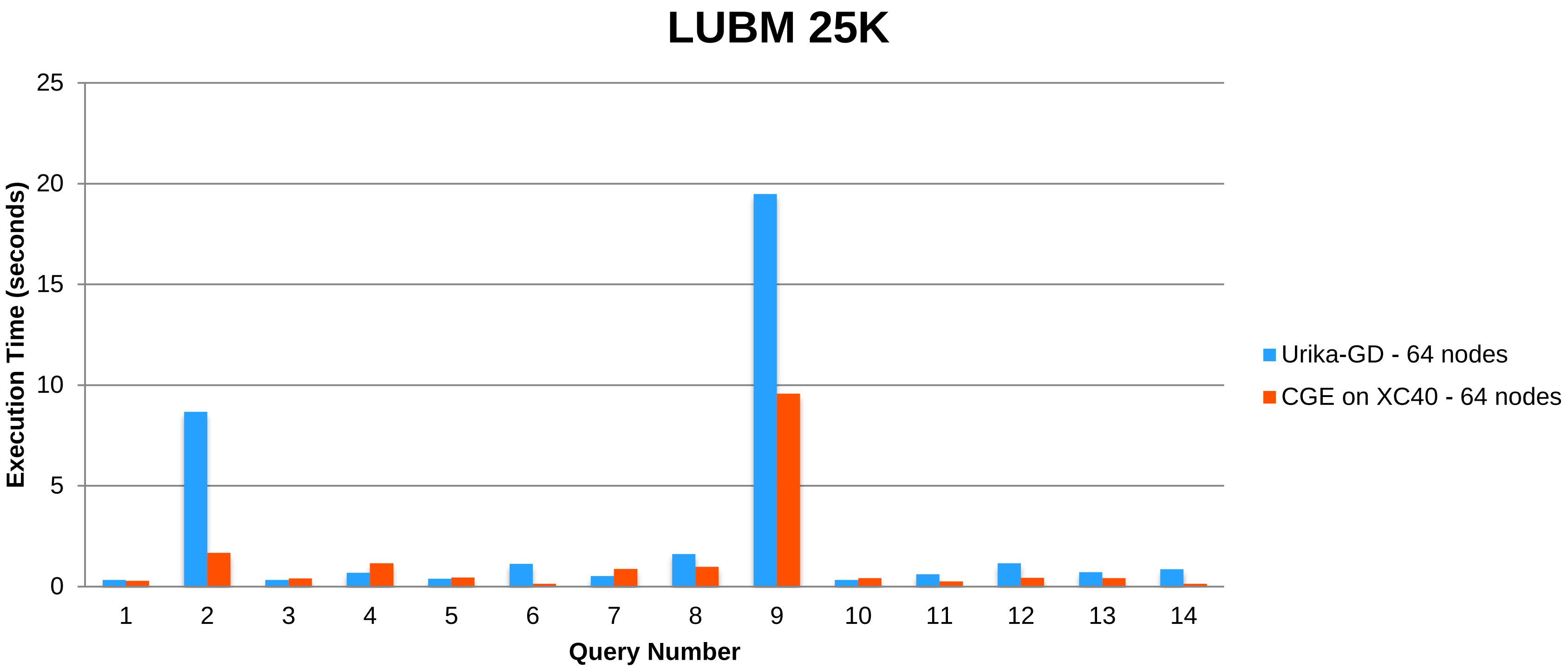
Performance on Standard Benchmarks

- **LUBM - Lehigh University BenchMark**
  - Concentrates primarily on the Basic Graph Pattern (BGP)
  - Communication intensive – tests parallel efficiency
  - LUBM25K is a moderate-sized benchmark (3 billion quads)
  - LUBM100K – 4x larger

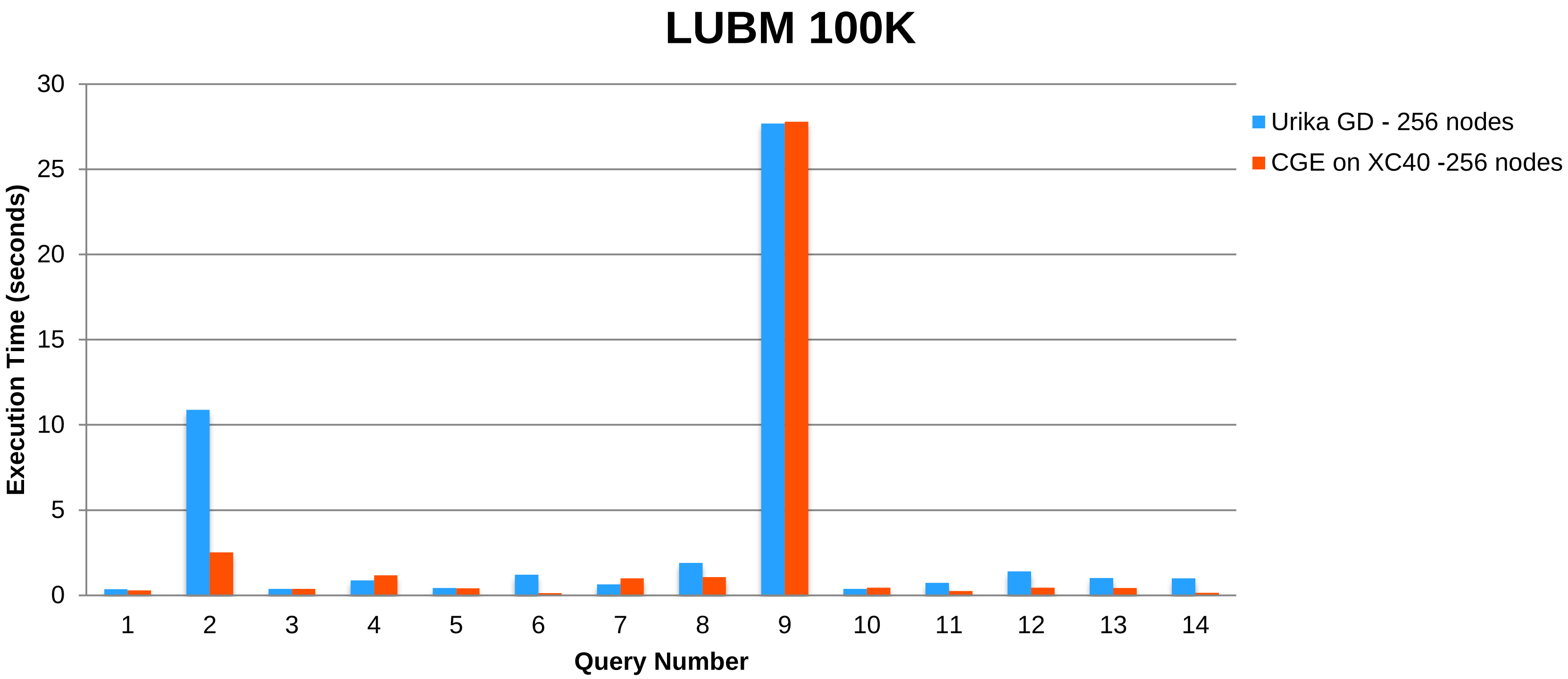
- **SP2B - SParql Performance Benchmark**
  - More dependent on the other SPARQL operators such as FILTER, DISTINCT and ORDER BY
  - Less sensitive to communications and more sensitive to node compute power

- **Test setup**
  - Load database and run all queries
  - Time reported is time within query engine for each query
  - LUBM25K tested on 64 node systems
  - LUBM100K and SP2B500M tested on 256 nodes

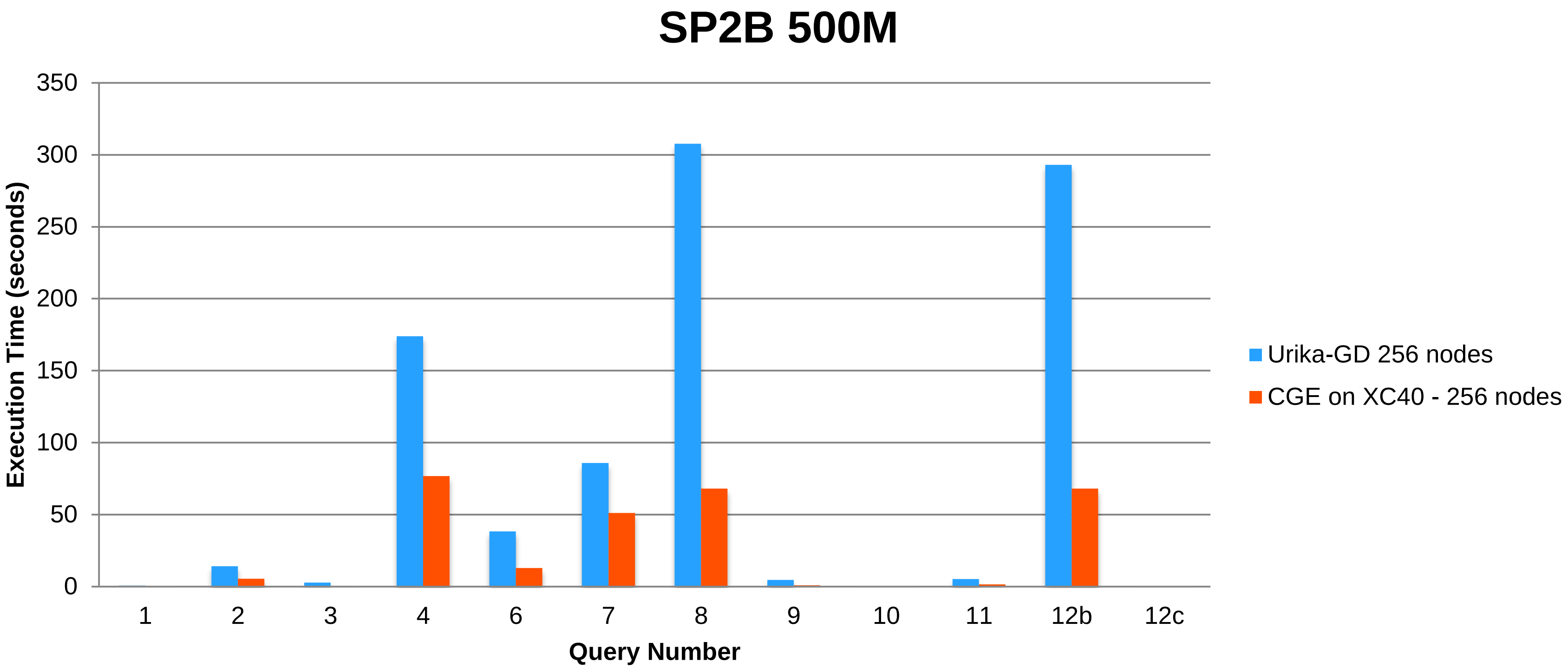
LUBM25K, 64 nodes



LUBM100K, 256 nodes



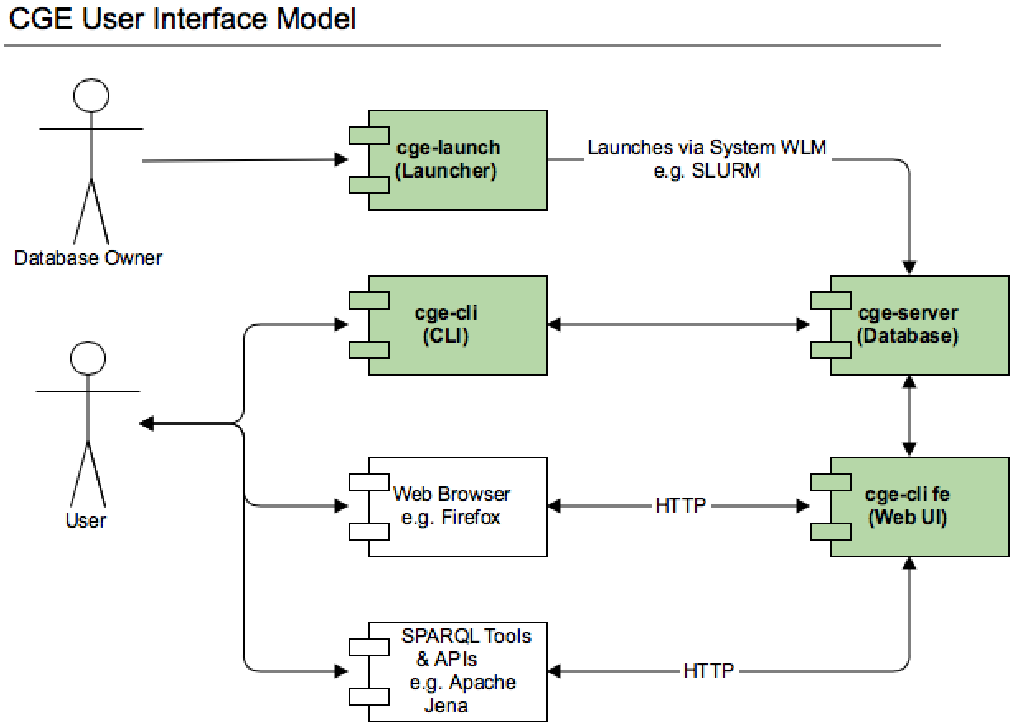
SP2B500M, 256 nodes



Conclusions from Benchmarking

- **Customers transitioning from Urika-GD will in general see improved performance on XC**
  - Based on comparison of equal number of nodes
    - Equal number of network injection ports
- **Even on the most complex query (LUBM Q9), CGE on XC is able to match the performance of Urika-GD**
  - Additional optimizations to improve scaling are in development
- **For simpler queries or queries which make use of compute intensive SPARQL operators, CGE on XC really shines**
  - Benefits from powerful compute nodes
- **Potential to scale to much larger node counts on XC than is possible using Urika-GD hardware as the back-end**

User Interface Model



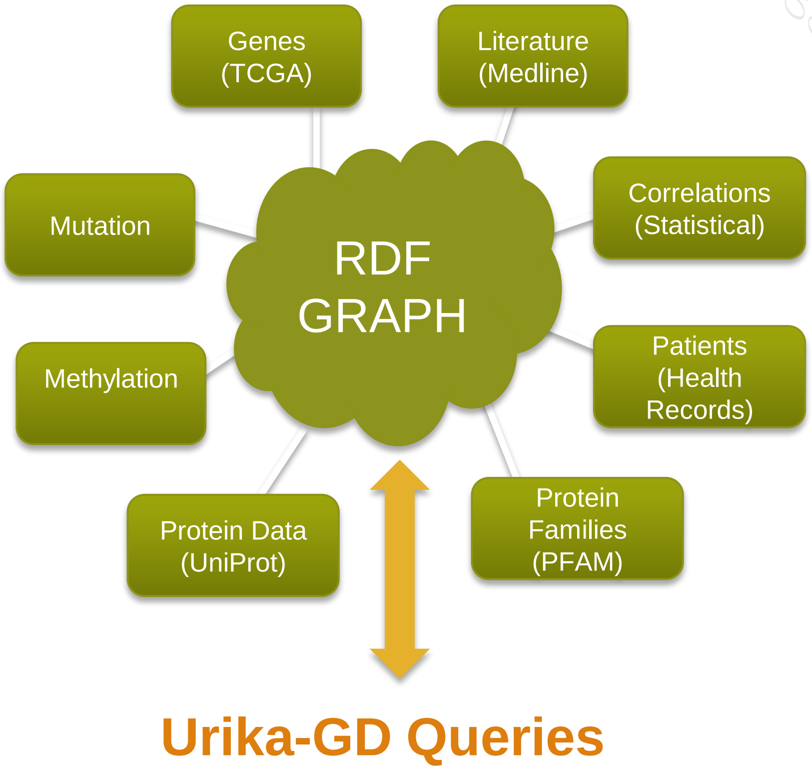
- Database owner launches the database server
- Users interact via their preferred interface
  - Command Line
  - Web Browser
  - SPARQL Tools & APIs
- CLI may be used for scripted workflows

Urika-GD Bioinformatics Use Case

Institute for Systems Biology (ISB) cancer drug repurposing study

- Multiple compute-intensive steps
- Multiple compute platforms
- Extensive data motion

Now could all be performed in a single platform!



Conclusions

- The Urika-GD Graph Analytic Database has been successfully ported from the Threadstorm architecture to the XC30/40, with good preliminary performance
- The Coarray C++ programming model provided a bridge from a shared memory architecture to distributed memory
- An interactive user interface and data security were retained, despite the differences in platform
- This exciting new capability will enable multi-step and mixed analytics workflows on a single platform
- For more information on this capability, contact Jim Maltby, jmaltby@cray.com

Thank You!

kristyn@cray.com
jmaltby@cray.com
rvesse@cray.com

Backup Slides

PGAS Porting Recommendations

Issue	Recommendation
Improve performance of remote PUTs and GETs	Disable SW Ordering by setting the environment variable setenv PGAS_NO_SW_ORDERING 1 Any ordering constraints must be handled explicitly by the user.
Increase concurrency for remote GETs	Issue multiple non-blocking operations before issuing an atomic_image_fence
Expose concurrency for remote GETs in deeply nested calls	Using lightweight threading layer to increase the number of non-blocking GETs issued
Reduce latency due to waiting on GET requests to complete	Overlap computation and communication

PGAS Porting Recommendations (continued)

Issue	Recommendation
Use of asymmetric coarrays incur additional overhead for remote address lookup.	When possible, use symmetric coarray allocations.
Improve performance of remote GETs.	Cache coptrs to remote memory.
DMAPP resource errors. Running out of resources for registering memory.	Explicitly register memory with PGAS. Currently using internal PGAS library registration function.
Memory fragmentation using tcmalloc. Prevents CGE from running lots of queries against a loaded database.	Preallocate memory to be managed by the application, using a custom buddy allocator.