Urika®-XC Analytic Applications Guide

(1.0UP00) S-2589

Contents

1 About the Urika®-XC Analytic Applications Guide (S-2589)

The *Urika®-XC Analytic Applications Guide (S-2589)* provides information about the features and analytic software components of Urika-XC software, as well instructions for using the analytic components.

Table 1. Record of Revision

Publication Title	Date	Release
Urika®-XC Analytic Applications Guide (S-2589)	August 2017	1.0UP00 release

Scope and Audience

This publication is written for users and administrators of Urika-XC.

Typographic Conventions

`Monospace`	Indicates program code, reserved words, library functions, command-line prompts, screen output, file/path names, and other software constructs.
`Monospaced Bold`	Indicates commands that must be entered on a command line or in response to an interactive prompt.
`Oblique` or *`Italics`*	Indicates user-supplied values in commands or syntax definitions.
Proportional Bold	Indicates a **GUI Window**, **GUI element**, cascading menu (**Ctrl**→**Alt**→**Delete**), or key strokes (press **Enter**).
\ (backslash)	At the end of a command line, indicates the Linux® shell line continuation character (lines joined by a backslash are parsed as a single line).

Trademarks

2 About Urika-XC

Cray Urika-XC is a high performance big data software stack, which is optimized for multiple work-flows and run on the Cray XC series systems. It features a comprehensive analytics software stack to help derive optimal business value from data. In addition, the software stack provides an optimized set of tools for capturing and organizing a wide variety of data types from different sources and for executing a variety of analytic jobs.

This release of Urika-XC software can be used with the CLE 6.0 UP02, CLE 6.0 UP03 and CLE 6.0 UP04 releases.

Features and Analytic Components

- **Cray Graph Engine (CGE)** - CGE is a highly optimized and scalable graph analytics application software, which is designed for high-speed processing of interconnected data. On Urika-XC, CGE jobs are scheduled like user applications, which is similar to the way other HPC applications are scheduled.
- **Open Source Analytics (OSA) images** - Urika-XC provides OSA images that run inside Shifter containers. Software provided in these images includes:
  - **Apache™ Spark™** - Spark is a general data processing framework that simplifies developing big data applications. It provides the means for executing batch, streaming, and interactive analytics jobs. In addition to the core Spark components, Urika-XC ships with a number of Spark ecosystem components.
  - **Anaconda® Python and R** - Anaconda is a distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing. It aims at simplifying package management and deployment.
  - **Dask and Dask Distributed** - Dask is a parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, data-frames, machine learning, and custom algorithms.
  - **Intel® BigDL** - BigDL is a distributed deep learning library for Spark that can run directly on top of existing Spark or Apache Hadoop clusters. Deep learning applications can be written as Scala or Python programs.

3 About the Cray Graph Engine (CGE)

CGE is a highly optimized software application designed for high-speed processing of interconnected data. It features an advanced platform for searching very large, graph-oriented databases and querying for complex relationships between data items in the database. It provides the tools required for capturing, organizing and analyzing large sets of interconnected data. CGE enables performing real-time analytics on the largest and most complex graph problems, and features highly optimized support for inference, deep graph analysis, and pattern-based queries.

3.1 CGE Features

Major features of CGE are listed below:

- An optimized query engine for high-speed parallel data analysis.
- Support for submitting queries, updates and creating checkpoints.
- A rich CLI.
- The CGE graphical user interface, which acts as a SPARQL 1.1 end point. This interface enables editing SPARQL queries or SPARUL updates and submitting them to the CGE database. It also accepts a set of commands that allow users to perform various tasks, such as creating a checkpoint on a database, setting Name Value Pairs (NVPs) to control certain aspects of data preprocessing, and query processing etc.
- SPARQL query language extension via the `INVOKE` and `PRODUCING` operators, which allow a classical graph algorithm to be passed an RDF graph and for the algorithm's results to be returned as data that is compatible with SPARQL 1.1. This enables graph algorithm library calls to be nested within a SPARQL query.
- Support for SPARQL aggregate functions.
- Multi-user support.
- Capability to cancel queries.
- Compatibility with POSIX-compliant file systems.
- Database preprocessing to apply inference rules to the data, as well as to index the data.
- CGE Python, CGE Java and CGE Spark APIs
- Support for a number of built in graph algorithms.

3.2 Get Started with Using CGE

About this task

This procedure can be used to get started with using CGE and can be considered as a "Hello World" program. In this procedure, a simple query is executed on a small RDF triples database. This procedure provides instructions for executing queries and viewing the results via the CGE CLI and the front end.

Use the `cge-cli help` command to view a full range of CGE CLI commands. Use the `-h` option of any command to view detailed help information about any specific command.

For a full set of CGE features, built in functions, graph algorithms, CGE API, troubleshooting and logging information, review the Cray Graph Engine (CGE) Users guide at *https://pubs.cray.com*.

Procedure

Authentication Setup

1. Set up SSH keys.

```
$ ssh localhost
```

If the preceding command allows re-logging into the login node without a password, then the SSH keys are set up sufficiently for using CGE. If the previous command fails and there are existing SSH keys that do not use pass-phrases or have the `ssh-agent` defined, then try the following

```
$ $ cat ~/.ssh/id_*.pub >> ~/.ssh/authorized_keys
```

At this point, if it is possible to run the aforementioned test and to re-log in to the login node without using a password, pass-phrase, or ssh-agent, then this step can be considered to be complete. On the other hand, if the aforementioned test fails, there are no SSH keys defined yet. The following commands can be used to set them up.

CAUTION: Before executing the following commands, ensure that there are no existing SSH keys because this will overwrite any existing keys. Also, do not specify a pass-phrase when running `ssh-keygen`

```
$ mkdir -p ~/.ssh
$ chmod 700 ~/.ssh
$ ssh-keygen
$ chmod 600 ~/.ssh/id_*
$ chmod 600 ~/.ssh/authorized_keys
```

Dataset Creation

2. Create a file named `dataset.nt` and store it in a directory that has been selected or created for it.

This directory must be a new directory and contain at least one of the following if the data set is being built for the first time with CGE (only one of these will actually be used):

- `dataset.nt` - This file contains triples and must be named dataset.nt
- `dataset.nq` - This file contains quads and must be named dataset.nq
- `graph.info` - This file contains a list of pathnames or URLs to files containing triples or quads and must be named `graph.info`.

This is the original, human readable representation of the database. The following example data can be used for this procedure.

```
<http://cray.com/example/spaceObject> <http://cray.com/example/hasName> "World" .
<http://cray.com/example/spaceObject> <http://cray.com/example/hasName> "Home Planet" .
<http://cray.com/example/spaceObject> <http://cray.com/example/hasName> "Earth" .
<http://cray.com/example/greeting> <http://cray.com/example/text> "Hello" .
<http://cray.com/example/greeting> <http://cray.com/example/text> "Hi"  .
```

Results Directory Creation and CGE Server Start-up

3. Select or create another directory into which the query engine should write the results and then launch the CGE server in a terminal window.

```
$ cge-launch -I 1 -N 1 -d /dirContainingExample/example -o \
/dirContainingExampleOutput -l :2
```

For more information about the `cge-launch` command and its parameters, see the `cge-launch` man page.

The server will output a few pages of log messages as it starts up and converts the database to its internal representation. When it finishes, the system will display a message similar to the following:

```
Serving queries on nid00057 16702
```

Query Execution via CGE CLI

4. Execute a query using the CGE CLI.

```
$ cge-cli query example.rq
0 [main] WARN com.cray.cge.cli.CgeCli  - User data hiding is enabled, logs will obscure/omit user
data.  Set cge.server.RevealUserDataInLogs=1 in the in-scope cge.properties file to disable this
behaviour.
5 [main] INFO com.cray.cge.cli.commands.queries.QueryCommand  - Received 1 queries to execute
13 [main] INFO com.cray.cge.cli.commands.queries.QueryCommand  - Running Query 1 of 1
0              6              123         0              file:///mnt/central/user/results/
queryResults.2017-07-04T13.59.57Z000.18232.tsv
688 [main] INFO com.cray.cge.cli.commands.queries.QueryCommand  - Query 1 of 1 succeeded
```

In the preceding example, the `example.rq` file contains the following query:

```
SELECT ?greeting ?object
WHERE
{
  <http://cray.com/example/greeting> <http://cray.com/example/test> ?
greeting .
  <http://cray.com/example/spaceObject> <http://cray.com/example/hasName> ?
object .
}
```

Use the following query to print just "Hello World" as the output:

```
SELECT ?greeting ?object
WHERE
{
  <http://cray.com/example/greeting> <http://cray.com/example/test> ?greeting .
  <http://cray.com/example/spaceObject> <http://cray.com/example/hasName> ?
object .
  FILTER(?greeting = "Hello" && ?object = "World")
}
```

Results Review

5. List the contents of the results directory and review the contents of the output file to verify that the query's results are stored in the output directory specified in the `cge-launch` command.

```
$ $ cd /dirContainingExampleOutput
$ ls
queryResults.34818.2015-10-05T19.33.53Z000.tsv
$ cat queryResults.34818.2015-10-05T19.33.53Z000.tsv
?greeting    ?object
"Hello"      "Home Planet"
"Hi"         "Home Planet"
"Hello"      "World"
"Hi"         "World"
"Hello"      "Earth"
"Hi"         "Earth"
```

CGE Front End Launch

6. Launch the CGE front end in another terminal window.

```
$ cge-cli fe --ping
```

The `--ping` option in the preceding example is used to verify that the database can be connected to immediately upon launch and that any failure is seen immediately. Not doing so may delay and hide failures. If the ping operation does not succeed, and it is certain that the user executing this command is the only user running CGE, and that everything else is set up correctly, the user should go back to the first step and make sure that the SSH keys are set up right. The system may prompt to trust the host key when the `fe` command is run for the first time.

Alternatively, the following command can be used to have the web server continue running in the background with its logs redirected, even if disconnected from the terminal session:

```
$ nohup cge-cli fe > web-server.log 2>&1 &
```

7. Point a browser at `http://machine-login1:3756` to launch web UI.

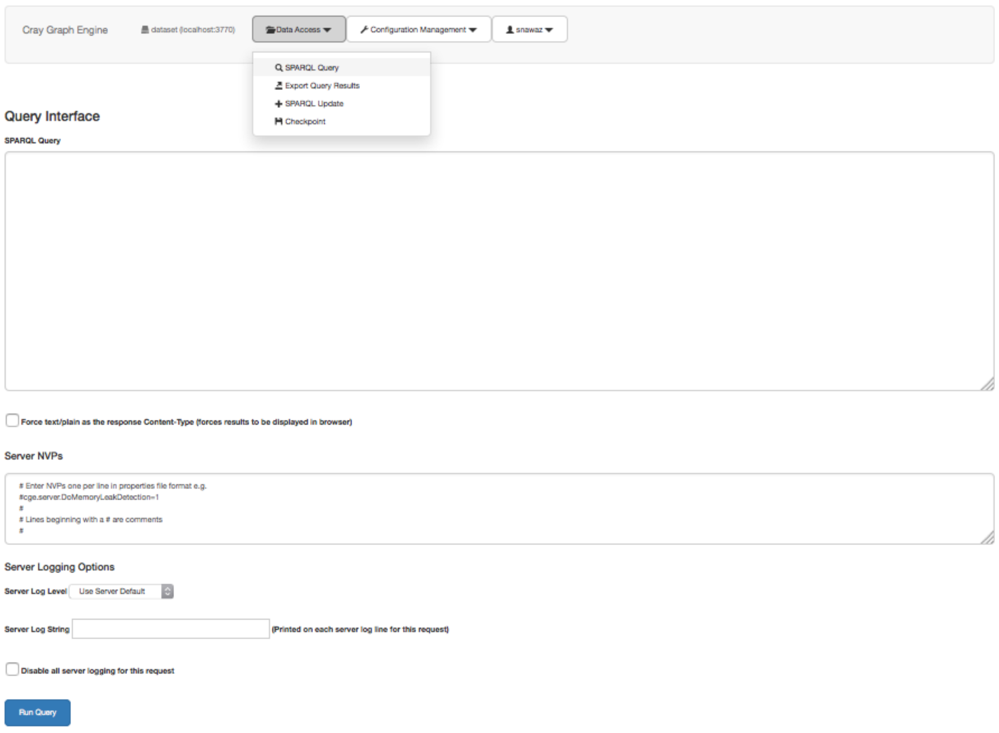
The CGE SPARQL protocol server listens at port `3756`, which is the default port ID.

When the CGE front end has been launched, a message similar to the following will be returned on the command-line:

```
49 [main] INFO com.cray.cge.cli.commands.sparql.ServerCommand -
CGE SPARQL Protocol Server has started and is ready to accept HTTP
requests on localhost:3756
```

Query Execution via the CGE Front End

8. Execute a query against the dataset created by typing in the query and selecting the **Run Query** button.



```
# Enter NVPs one per line in properties file format e.g.
#cge.server.DoMemoryLeakDetection=1
#
# Lines beginning with a # are comments
#
```

Server Log Level: Use Server Default

Figure 1. CGE Query Interface

The following example query will match the data and example output shown in the next step:

```
SELECT ?greeting ?object
WHERE
{
  <http://cray.com/example/greeting> <http://cray.com/example/test> ?
greeting .
  <http://cray.com/example/spaceObject> <http://cray.com/example/hasName> ?
object .
}
```

After the query finishes executing, the output file containing the query's results will be stored in the output directory that was specified in the `cge-launch` command.

CGE Front End Termination

9. Quit the terminal using the `CTRL+C` keyboard shortcut.

CGE Server Shutdown

10. Execute the following command to halt the CGE server, if needed.

```
$ cge-cli shutdown
```

4 About Open Source Analytics (OSA) Images

OSA images contain everything required for running Spark, Dask Distributed, Anaconda Python, and BigDL programs on the Urika-XC system. The `start_analytics` script creates and runs Shifter containers on allocated nodes of the XC system using OSA images.

Only OSA and CGE images can be used as part of Urika-XC software. Downloading additional images and integrating them into the Urika-XC software is not supported.

For more information, see the `start_analytics` man page.

4.1 Shifter System Introduction

The Shifter system allows users to run commands in a user defined image (UDI). A UDI has many of the features of a Linux container. Some features (e.g., separate pid namespace) are missing because the UDI is intended to be used on a compute node where only one user's apps run at a time.

A UDI is created from a Docker image. A user may download an image that has been exported by another site from the Docker Hub. This image can then be unpacked and then packaged into a UDI for the local user to use. Downloading and packaging is done with the `shifterimg` command.

Once a user has a UDI to use, the user can submit a batch job to use the image. The batch job can request use of the image in one of two ways:

1. When the job is submitted by specifying an environment variable that indicates what UDI should be used, or
2. By submitting a job using the `shifter` command inside the batch job.

Both of these methods are described in more detail in the Shifter User Guide.

Shifter on Urika-XC

Shifter allows users to provide a completely pre-packaged analytics environment with all the necessary dependencies. Users acquire an allocation of nodes from their systems workload manager/scheduler, and the Urika-XC start up script creates a cluster of Shifter containers on the allocated nodes, which are configured to talk to each other. Everything except for CGE runs in Shifter containers (i.e., all of the Open Source Analytics (OSA) components shipped with the Urika-XC). Shifter also provides the per-node cache functionality that creates a loop back mounted file system on every node. This provides efficient emulation of local storage for frameworks like Spark that require it.

4.2 About the `start_analytics` Command

The `start_analytics` command starts an analytics cluster, which can be used to run Open Source Analytics (OSA) components, including Spark, Anaconda, Dask, and BigDL. It can be considered as an entry point to Urika-XC's OSA components.

The `start_analytics` command also accepts options that enable users to:

- Run commands in the analytics cluster and exit, instead of opening an interactive shell.
- Start a Dask Distributed cluster.
- Launch Dask Distributed with the specified memory limit, desired number of workers and/or cores.
- Start a single analytics container on the current login node.
- Specify a Conda environment to start Dask workers and Dask Scheduler with.

For more information, see the `start_analytics` man page.

4.3 About Apache Spark

Apache™ Spark™ is a fast and general engine for data processing. It provides high-level APIs in Java, R, Scala and Python, and an optimized engine.

- **Spark Core, DataFrames, and Resilient Distributed Datasets (RDDs)** - Spark Core provides distributed task dispatching, scheduling, and basic I/O functionalities.
- **Spark SQL, DataSets, and DataFrames** - The Spark SQL component is a layer on top of Spark Core for processing structured data.
- **Spark Streaming** - The Spark Streaming component leverages Spark Core's fast scheduling capabilities to perform streaming analytics.
- **MLlib Machine Learning Library** - MLlib is a distributed machine learning framework on top of Spark.
- **GraphX** - GraphX is a distributed graph processing framework on top of Spark. It provides an API for expressing graph computations.

This section provides a quick guide to using Apache Spark. Please refer to the official Apache Spark documentation for detailed information about Spark, as well as documentation of the Spark APIs, programming model, and configuration parameters.

Urika-XC ships with Spark 2.1.1.

Run Spark Applications

The Urika-XC software stack includes Spark configured and deployed to run in a Shifter container, with a per-node cache for local temporary storage.

To launch Spark applications or interactive shells, use the standard Spark launch scripts from the interactive container that is created when an analytics cluster is launched using `start_analytics`. These scripts include:

- `spark-shell`
- `spark-submit`
- `spark-sql`

- `pyspark`
- `sparkR`
- `run-example`

The Spark startup scripts will by default start up a Spark instance across all cores in the allocation.. To request a smaller or larger instance, pass the *`--total-executor-cores No_of_Desired_cores`* command-line flag. Memory allocated to Spark executors and drivers can be controlled with the `--driver-memory` and `--executor-memory` flags. By default, 32 Gigabytes are allocated to the driver, and 32 Gigabytes are allocated to each executor, but this will be overridden if a different value is specified via the command-line, or if a property file is used.

Further details about starting and running Spark applications are available at *http://spark.apache.org*

Build Spark Applications

Spark 2.1.1 builds with Scala 2.11.8. Urika-XC ships with Maven installed for building Java applications (including applications utilizing Spark's Java APIs), and Scala Build Tool (sbt) for building Scala Applications (including applications using Spark's Scala APIs). To build a Spark application with these tools, add a dependence on Spark to the build file. For Scala applications built with `sbt`, add this dependence to the `.sbt` file, such as in the following example:

```
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.1"
```

For Java applications built with Maven, add the necessary dependence to the `pom.xml` file, such as in the following example :

```
<dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.1.1</version>
    </dependency>
</dependencies>
```

For detailed information on building Spark applications, please refer to the current version of Spark's programming guide at *http://spark.apache.org*.

Spark Configuration Differences on Urika-XC

Spark's default configurations on Urika-XC have a few differences from the standard Spark configuration:

- **Changes to improve execution over a high-speed interconnect** - The presence of the high-speed Aries network on Urika-XC changes some of the tradeoffs between compute time and communication time. Because of this, the default settings of `spark.shuffle.compress` has been changed to `false` and that of `spark.locality.wait` has been changed to 1. This results in improved execution times for some applications on Urika-XC.
- **Increases to default memory allocation** - Spark's standard default memory allocation is 1 Gigabyte to each executor, and 1 Gigabyte to the driver. Due to large memory nodes, these defaults were changed to 32 Gigabytes for each executor and 32 Gigabytes for the driver.
- **Local temporary cache** - Spark on Urika-XC is configured to utilize a per node loopback filesystem provided by Shifter for it's local temporary storage.

Conda Environments

PySpark on Urika-XC is aware of Conda environments. If there is an active Conda environment (the name of the environment is prepended to the Unix shell prompt), the PySpark shell will detect and utilize the environment's Python. To override this behavior, manually set the `PYSPARK_PYTHON` environment variable to point to the preferred Python. For more information, see *Enable Anaconda Python and the Conda Environment Manager* on page 14.

4.3.1 Start Up Apache Spark and the Analytic Programming Environment

Prerequisites

This procedure assumes that the workload manager being used is either Slurm or Moab Torque.

About this task

Urika-XC allows users to create their own analytics clusters on a set of nodes allocated from Slurm or Moab Torque. Once created, this cluster also contains Anaconda Python, BigDL, and optionally Dask Distributed, if it was started with the appropriate options.

Procedure

1. Load the `analytics` module.

   ```
   $ module load analytics
   ```

2. Optional: Set values for environment variables.

   Certain environment variables may be set before running the `start_analytics` script to modify the behavior of the analytics cluster. Setting values for these variables is optional. Furthermore, these variables have reasonable default values.

   - `MINERVA_USE_LOGIN` - If this environment variable is set, the interactive shell will run on the login rather than a compute node. This allows better external connectivity for build and environment tools that need to download new packages.
   - `SPARK_LOOPBACK_SIZE` - Sets the size of the per-node loopback mounted local file system used by Spark for local storage. The default value of this variable is 256 GB.
   - `SPARK_EVENT_DIR` - Sets the location for Spark event logs.

   For more information, see the `start_analytics` man page.

3. Allocate the desired number of nodes in the `interactive` mode and execute the `start_analytics` script.

   If *N* number of nodes are allocated, one of them will be allocated as a master and one of them will be allocated as an interactive node. In addition:

   - If the system uses Moab Torque, *N*-1 worker containers will be launched, because the interactive container is always launched on the login node with Moab Torque.
   - If the system uses Slurm, *N*-2 worker containers will be launched.

   For example, to run a cluster with 16 worker nodes, execute the following command:

   Example for Slurm:

```
$ salloc -N 18 start_analytics
```

Example for Moab Torque:

```
$ qsub -I -l nodes=17
$ start_analytics
```

The `start_analytics` script features the `-d` option that starts a single analytics container on the current login node. No job allocation is required. Spark can still be used in local mode. This is useful for performing development work, such as creating Conda environments, building applications, running single node tests etc. In addition, the `-d` option enables performing development tasks with full access to the analytics environment, without having to wait for a job allocation. Since this option provides better access to the external network, it can be useful for downloading new packages for builds.

Executing the `start_analytics` script presents a Bash shell on one of the cluster nodes, where Spark and/or the analytic programming environment commands can be executed.

4.4 Enable Anaconda Python and the Conda Environment Manager

About this task

Urika-XC Spark and the analytic application programming environment clusters come with the Anaconda Python distribution version 4.4.0, including the Conda package and environment manager. This is the recommended Python distribution for running analytic jobs using Urika-XC. If there is an active Conda environment, PySpark will automatically utilize Anaconda.

Procedure

1. Start an analytics cluster.

   This will place the user on a node running an interactive container. `nid00030` is used as an example for an interactive container node in this procedure.

2. Create a Conda environment.

   The following example creates a Conda environment with `scipy` and all of its dependencies loaded:

   ```
   [user@nid00030 ~]$ conda create --name scipyEnv scipy
   ```

   Use the `conda config --add envs_dirs` *`path_to_directory`* command if it is required to set an alternate environments directory for Conda. *`path_to_directory`* must be a directory that is mounted within the container.

3. Activate the Conda environment.

   ```
   [user@nid00030 ~]$ source activate scipyEnv
   ```

   For more information about Anaconda, please refer to *https://docs.continuum.io*. For additional information about the Conda environment manager, please refer to *http://conda.pydata.org/docs/*

4.5 Dask on Urika-XC

Dask is a Python based parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, data-frames, machine learning, and custom algorithms. The Dask package for Python is a library for parallel analytics computing. It supports multiple styles of task scheduling, as well as multiple parallel data structures. The Dask distributed package for Python is a distributed scheduler that allows Dask computations to be parallelized across multiple nodes. Distributed requires starting up a single scheduler process, in addition to one or more worker processes.

To learn more about Dask, visit *http://dask.pydata.org/en/latest/*, *https://dask.pydata.org/* and *https://distributed.readthedocs.io/*.

Dask on Urika-XC is supported with Anaconda Python versions 2.7, 3.5, and 3.6. It is currently not supported with Python 3.4 as this version of Python does not support the Dask Scheduler files that Urika-XC uses to coordinate workers with the Client and Scheduler.

Urika-XC automatically sets up Dask Distributed in the analytics cluster if `start_analytics` is executed with certain options. For more information, see the `start_analytics` man page.

4.5.1 Use Dask to Run Python Programs

Prerequisites

This procedure requires Urika-XC software installed on the system.

About this task

Procedure

1. Create a Conda environment with the Dask and Distributed packages, and any other Python packages and versions to use with Dask.

   This can be done in development mode.

   ```
   $ ./start_analytics_dask_dev -d
   bash-4.2$ conda create --name mydaskenv dask distributed biopython python=3.5
   bash-4.2$ conda info --envs
   conda environments:
   mydaskenv                /home/users/name/.conda/envs/mydaskenv
   bash-4.2$ exit
   ```

2. Start an analytics cluster, using the `--dask` or `-k` option to start Dask, and the `--dask-env` or `-e` option to specify the conda environment.

   ```
   $ salloc -N 40 ./start_analytics_dask_dev -k -e mydaskenv
   Analytics cluster ready.  Type 'spark-shell' for an interactive Spark shell.
   (mydaskenv)
   ```

3. Run a Python program or start an interactive REPL.

To use Dask Distributed while running a Python program, specify the scheduler file location when initializing the client. The scheduler file location can be found in `$DASK_SCHED_FILE`.

```
(mydaskenv) python
Python 3.5.3 |Continuum Analytics, Inc.| (default, Mar  6 2017, 11:58:13)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> from dask import bag
>>> from distributed import Client
>>> client = Client(scheduler_file=os.environ['DASK_SCHED_FILE'])
>>>
```

4.6 About Intel BigDL

The BigDL distributed deep learning library was developed for Apache Spark and is targeted at Spark users who want to apply deep learning to data already available through Spark. BigDL also allows users to develop and run deep learning applications from within Spark. BigDL leverages Spark to efficiently scale-out BigDL to run across multiple nodes, but can also be run on a single node as a local Java or Scala program.

BigDL is modeled after Torch and provides support for adding deep learning (both training and inference) to Spark applications and workflows. Users can also load pre-trained Caffe or Torch models into Spark programs using BigDL.

For more information, visit *https://bigdl-project.github.io/0.1.1/index.html* to review to the '*Programming Guide for BigDL*', which covers BigDL concepts and APIs for building deep learning applications.

BigDL on Urika-XC

BigDL is built with MKL support and is pre-installed on Urika-XC. The BigDL distribution package is located under `/opt/BigDL/dist` on the Urika-XC system. The version of BigDL used on Urika-XC is 0.1.1

Use the following environment variables to run a deep learning tasks with the BigDL toolkit:

- `BIGDL_DIR`: Carries the location of the BigDL files necessary to set up the environment and attach the proper configuration and JAR files
- `BIGDL_JAR`: Carries the location of the BigDL JAR file to be used when starting a Spark shell.

4.6.1 Run BigDL Programs Using spark-submit

Prerequisites

This procedure assumes that the workload manager being used is either Slurm or Moab Torque.

About this task

BigDL uses the Intel MKL library to achieve high performance. To efficiently use the MKL routines, some environment variables need to be set via the `$BIGDL_DIR/scripts/bigdl.sh` script.

In the following procedure, the `bigdl.sh` script is used to launch `spark-submit` or `spark-shell` option to run the LeNet5 training example using the MNIST dataset.

Procedure

1. Log on to a login node.
2. Start up Spark and the analytics programming environment.
   a. Load the `analytics` module.

   ```
   $ module load analytics
   ```

   b. Set values for environment variables.

   Certain environment variables may be set before running the `start_analytics` script to modify the behavior of the analytics cluster. Setting values for these variables is optional. Furthermore, these variables have reasonable default values.

   - `MINERVA_USE_LOGIN` - If this environment variable is set, the interactive shell will run on the login rather than a compute node. This allows better external connectivity for build and environment tools that need to download new packages.
   - `SPARK_LOOPBACK_SIZE` - Sets the size of the per-node loopback mounted local file system used by Spark for local storage. The default value of this variable is 256 GB.
   - `SPARK_EVENT_DIR` - Sets the location for Spark event logs.

   For more information, see the `start_analytics` man page.

   c. Allocate the desired number of nodes in the `interactive` mode and execute the `start_analytics` script.

   If *N* number of nodes are allocated, one of them will be allocated as a master and one of them will be allocated as an interactive node. In addition:

   - If the system uses Moab Torque, *N*-1 worker containers will be launched, because the interactive container is always launched on the login node with Moab Torque.
   - If the system uses Slurm, *N*-2 worker containers will be launched.

   The `start_analytics` script features the `-d` option that starts a single analytics container on the current login node. No job allocation is required. Spark can still be used in local mode. This is useful for performing development work, such as creating Conda environments, building applications, running single node tests etc. In addition, the `-d` option enables performing development tasks with full access to the analytics environment, without having to wait for a job allocation. Since this option provides better access to the external network, it can be useful for downloading new packages for builds.

   Executing the `start_analytics` script presents a Bash shell on one of the cluster nodes, where Spark and/or the analytic programming environment commands can be executed.

3. Set the required environment variables using one of the following mechanisms:

   - Source the `$BIGDL_DIR/bin/bigdl.sh` file to set the required environment variables globally and then launch the program.

   ```
   $ source $BIGDL_DIR/bin/bigdl.sh
   $ spark-submit --total-executor-cores 640 \
   --conf spark.executor.instances=32 --conf spark.executor.cores=20 \
   --conf spark.shuffle.reduceLocality.enabled=false \
   --class com.intel.analytics.bigdl.models.lenet.Train \
   $BIGDL_DIR/lib/bigdl-0.1.1-jar-with-dependencies.jar \
   ```

```
-f /dir/username/mnist -b $batch_size -r 0.10 \
--checkpoint ./tests/log/model
```

- Execute the `$BIGDL_DIR/bin/bigdl.sh` script with the `spark-shell` or `spark-submit` option to set the required environment variables for the duration of the session and launch the program.

```
$BIGDL_DIR/bin/bigdl.sh -- spark-submit --total-executor-cores 640 \
--conf spark.executor.instances=32 --conf spark.executor.cores=20 \
--conf spark.shuffle.reduceLocality.enabled=false \
--class com.intel.analytics.bigdl.models.lenet.Train \
$BIGDL_DIR/lib/bigdl-0.1.1-jar-with-dependencies.jar \
-f /dir/username/mnist -b $batch_size -r 0.10 \
--checkpoint ./tests/log/model
```

The parameters used in the preceding examples include:

- `-f`: Specifies where the MNIST data is placed.
- `--checkpoint`: Specifies where the `model`/`train_state` snapshot can be cached. Input a folder and ensure the folder is created this example is run. The model snapshot will be named as `model.`*`#iteration_number`*, and train state will be named as `state.`*`#iteration_number`*. If there are any files already existing in the folder, the old file(s) will not be overwritten for the safety of model files.
- `-b`: Specifies the mini-batch size. It is expected that the mini-batch size is a multiple of *`node_number`*`*` *`core_number`*.

4.6.2 Run BigDL Programs Using PySpark

Prerequisites

This procedure assumes that the workload manager being used is either Slurm or Moab Torque.

About this task

This procedure enables users to run PySpark applications on Urika-XC images using BigDL. In the following procedure, the `bigdl.sh` script is used with the `spark-submit` and `spark-shell` options for executing the `Textclassification` example with the GloVe and News20 datasets. The text classification test requires the GloVe (Global vectors for Word Representation) dataset, which is approximately 823 MB. Since job allocation may timeout if this dataset is downloaded at runtime, the dataset should be downloaded before running any tests. The tests need to be modified to access datasets from a local directory. To modify the text classification example, change the function calls in `textclassification.py` from:

```
news20.get_news20()
new20.get_glove_w2(dim=embedding_dim)
```

to:

```
news20.get_news20(source_dir=”pathto/dataset”)
news20.get_glove_w2v(source_dir=”pathto/dataset”,dim=embedding_dim)
```

Procedure

1. Log on to a login node.

2. Start up Spark and the analytics programming environment.

   a. Load the `analytics` module.

   ```
   $ module load analytics
   ```

   b. Set values for environment variables.

   Certain environment variables may be set before running the `start_analytics` script to modify the behavior of the analytics cluster. Setting values for these variables is optional. Furthermore, these variables have reasonable default values.

   - `MINERVA_USE_LOGIN` - If this environment variable is set, the interactive shell will run on the login rather than a compute node. This allows better external connectivity for build and environment tools that need to download new packages.
   - `SPARK_LOOPBACK_SIZE` - Sets the size of the per-node loopback mounted local file system used by Spark for local storage. The default value of this variable is 256 GB.
   - `SPARK_EVENT_DIR` - Sets the location for Spark event logs.

   For more information, see the `start_analytics` man page.

   c. Allocate the desired number of nodes in the `interactive` mode and execute the `start_analytics` script.

   If *N* number of nodes are allocated, one of them will be allocated as a master and one of them will be allocated as an interactive node. In addition:

   - If the system uses Moab Torque, *N*-1 worker containers will be launched, because the interactive container is always launched on the login node with Moab Torque.
   - If the system uses Slurm, *N*-2 worker containers will be launched.

   The `start_analytics` script features the `-d` option that starts a single analytics container on the current login node. No job allocation is required. Spark can still be used in local mode. This is useful for performing development work, such as creating Conda environments, building applications, running single node tests etc. In addition, the `-d` option enables performing development tasks with full access to the analytics environment, without having to wait for a job allocation. Since this option provides better access to the external network, it can be useful for downloading new packages for builds.

   Executing the `start_analytics` script presents a Bash shell on one of the cluster nodes, where Spark and/or the analytic programming environment commands can be executed.

3. Set the root environment to access Spark libraries.

   ```
   $ source activate root
   ```

4. Create a variable for Python libraries.

   ```
   $ export PYTHON_API_ZIP_PATH=${BIGDL_DIR}/lib/bigdl-0.1.1-python-api.zip
   ```

5. Set the Python path.

   ```
   $ export PYTHONPATH=${PYTHON_API_ZIP_PATH}:$PYTHONPATH
   ```

6. Source the BigDL environment and use `spark-submit` to execute the `pyspark` test.

   ```
   $ source $BIGDL_DIR/bin/bigdl.sh
   $ spark-submit --total-executor-cores 80 \
   ```

```
 --conf spark.executor.instances=4 --conf spark.executor.cores=20 \
--py-files ${PYTHON_API_ZIP_PATH},
/pathto/textclassifier.py  \
 --jars ${BIGDL_JAR} \
 --conf spark.executorEnv.PYTHONHASHSEED=123 \
 /pathto/textclassifier.py  -b 320 --max_epoch 3 --model cnn
```

7. Execute the `bigdl.sh` script with the `spark-submit` option to set the required environment variables for the duration of the session and launching of the program.

   In the following example, `-b` specifies the mini-batch size. It is expected that the mini-batch size is a multiple of *`node_number`* * *`core_number`*.

```
$ BIGDL_DIR/bin/bigdl.sh -- spark-submit --total-executor-cores 80 \
 --conf spark.executor.instances=4 --conf spark.executor.cores=20 \
 --py-files ${PYTHON_API_ZIP_PATH},
/pathto/textclassifier.py  \
 --jars ${BIGDL_JAR} \
 --conf spark.executorEnv.PYTHONHASHSEED=123 \
 /pathto/textclassifier.py  -b 320 --max_epoch 3 --model cnn
```

4.6.3 Get Started with BigDL

BigDL programs can be executed after launching a Spark shell. Use the following methods to get familiar with using BigDL for performing deep learning tasks:

- Run `spark-shell` with BigDL.

```
$ source $BIGDL_DIR/bin/bigdl.sh
$ spark-shell --properties-file $BIGDL_DIR/conf/spark-bigdl.conf --jars $BIGDL_JAR
```

- Use the BigDL Tensor API.

```
scala> import com.intel.analytics.bigdl.tensor.Tensor
import com.intel.analytics.bigdl.tensor.Tensor
scala> Tensor[Double](2,2).fill(1.0)
res0: com.intel.analytics.bigdl.tensor.Tensor[Double] =
1.0          1.0
1.0          1.0
[com.intel.analytics.bigdl.tensor.DenseTensor of size 2x2]
```

- Use the LeNet on MNIST "Hello World" deep learning example, which trains LeNet-5 on the MNIST data using BigDL. For more information, visit *https://bigdl-project.github.io/0.1.1/#UserGuide/examples/* and see '*Training LeNet on MNIST - The "hello world" for deep learning*'. The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.
- Build complex deep learning models and applications using BigDL. examples accessible at *https://bigdl-project.github.io/0.1.1/#UserGuide/examples/*. These examples are pre-built with the BigDL distribution and demonstrate how to use BigDL to train and evaluate several of the supported neural network models. Use the following bash script to call one of these pre-built examples:

```
 # Launch BigDL job
 function launchBigDLJob() {
  # echo "Entering function: launchBigDL"
  local worker_nodes=`expr $SLURM_JOB_NUM_NODES - 2`
  local cores=`expr $worker_nodes '*' 20`
  local batch_size=`expr $cores '*' 4`
  echo "Number of Worker_nodes $worker_nodes"
  echo "Running BigDL LeNet5 training with $cores cores with batch size
 $batch_size"

 $BIGDL_DIR/bin/bigdl.sh -- spark-submit --total-executor-cores $cores \
 --conf spark.executor.instances=$worker_nodes --conf spark.executor.cores=20 \
 --conf spark.shuffle.reduceLocality.enabled=false \
 --class com.intel.analytics.bigdl.models.lenet.Train \
 $BIGDL_DIR/lib/bigdl-0.1.1-jar-with-dependencies.jar \
 -f /lus/snx11254/kristyn/mnist -b $batch_size -r 0.10 \
 --checkpoint ./tests/log/model
  # echo "Exiting function: launchBigDLJob"
 }
```

4.6.4 Run Big Programs as Local Java or Scala Programs

Prerequisites

This procedure assumes that the workload manager being used is either Slurm or Moab Torque.

About this task

BigDL can also be run on a single node as a local Java or Scala program outside of Spark.

Procedure

1. Start up Spark and the analytics programming environment.

a. Load the `analytics` module.

```
$ module load analytics
```

b. Set values for environment variables.

Certain environment variables may be set before running the `start_analytics` script to modify the behavior of the analytics cluster. Setting values for these variables is optional. Furthermore, these variables have reasonable default values.

- `MINERVA_USE_LOGIN` - If this environment variable is set, the interactive shell will run on the login rather than a compute node. This allows better external connectivity for build and environment tools that need to download new packages.
- `SPARK_LOOPBACK_SIZE` - Sets the size of the per-node loopback mounted local file system used by Spark for local storage. The default value of this variable is 256 GB.
- `SPARK_EVENT_DIR` - Sets the location for Spark event logs.

For more information, see the `start_analytics` man page.

c. Allocate the desired number of nodes in the `interactive` mode and execute the `start_analytics` script.

If *N* number of nodes are allocated, one of them will be allocated as a master and one of them will be allocated as an interactive node. In addition:

- If the system uses Moab Torque, *N*-1 worker containers will be launched, because the interactive container is always launched on the login node with Moab Torque.

- If the system uses Slurm, *N*-2 worker containers will be launched.

  The `start_analytics` script features the `-d` option that starts a single analytics container on the current login node. No job allocation is required. Spark can still be used in local mode. This is useful for performing development work, such as creating Conda environments, building applications, running single node tests etc. In addition, the `-d` option enables performing development tasks with full access to the analytics environment, without having to wait for a job allocation. Since this option provides better access to the external network, it can be useful for downloading new packages for builds.

  Executing the `start_analytics` script presents a Bash shell on one of the cluster nodes, where Spark and/or the analytic programming environment commands can be executed.

2. Set `DL_CORE_NUMBER` to the desired number of cores and set `BIGDL_LOCAL_MODE` to `true` to indicate that BigDL needs to run locally/outside of Spark.

```
$ source $BIGDL_DIR/bin/bigdl.sh
$ export BIGDL_LOCAL_MODE=true
$ export DL_CORE_NUMBER=8
$ scala -cp  my_bigdltests_2.11-1.0.jar:$BIGDL_JAR  MyLeNetTrainLocal -f \
/lus/scratch/datasets/mnist
```

Depending on the language, use the following format for executing this code:

- Java:

```
java -cp fileName.jar nameOfUsersmainClass
```

- Scala:

```
scala -cp fileName.jar nameOfUsersmainClass
```

In the preceding examples, *`fileName`* represents the name of JAR file(s) containing the user's `main` class, as well as all the associated dependencies.

WARNING:

The mechanism described above does not work on post BigDL 0.1.1 releases.

4.6.5 Intel BigDL Logging

BigDL implements a method named `redirectSparkInfoLogs`, which is is is used in many BigDL examples to redirect logs of `org`, `akka`, and `breeze` to `bigdl.log` with a log setting of `INFO`, except `org.apache.spark.SparkContext`. This method returns error messages to the console. By default, the `bigdl.log` log file will be generated under the current directory or workspace that `spark-submit` is launched from.

The following import and call to `redirectSparkInfoLogs()` will be seen in the example codes.

```
import com.intel.analytics.bigdl.utils.LoggerFilter
LoggerFilter.redirectSparkInfoLogs()
```

Set the value of the `-Dbigdl.utils.LoggerFilter.disable` Java property to `true` to disable the redirection of these logs to `bigdl.log`, as shown in the following example:

```
-Dbigdl.utils.LoggerFilter.disable=true
```

By default, all the examples and models in the code will be redirected. Specify where the `bigdl.log` file will be generated by setting the value of the `Dbigdl.utils.LoggerFilter.logFile` parameter to the desired location, as shown in the following example:

```
Dbigdl.utils.LoggerFilter.logFile=path
```

By default, it will be generated under current workspace. Extra Java properties are passed into `spark-submit` using the `spark.driver.extraJavaOptions` and `spark.executor.extraJavaOptions` configuration parameters.

For example, to run the LeNet5 Training example and have the `bigdl.log` file stored in a different directory than the current working directory, include the `--conf spark.driver.extraJavaOptions="-Dbigdl.utils.LoggerFilter.logFile=/lus/scratch/my_bigdl_logs/bigdl.log"` setting, as shown in the following example:

```
$BIGDL_DIR/bin/bigdl.sh -- spark-submit --total-executor-cores 640 \
--conf spark.executor.instances=32 --conf spark.executor.cores=20 \
--conf spark.shuffle.reduceLocality.enabled=false \
--conf spark.driver.extraJavaOptions="-Dbigdl.utils.LoggerFilter.logFile=\/lus/scratch/my_bigdl_logs/bigdl.log" \
--class com.intel.analytics.bigdl.models.lenet.Train $BIGDL_DIR/lib/bigdl-0.1.1-jar-with-dependencies.jar \
-f /lus/snx11254/kristyn/mnist -b 2560 -r 0.10 --checkpoint ./tests/log/model
```

Use logging messages to easily track the `epoch/iteration/loss/throughput` directly from the log file when running Training with BigDL.

For example use the `grep Epoch bigdl.log` or `grep Iteration bigdl.log` commands to monitor training progress. Similarly, use the `grep Accuracy bigdl.log` command to monitor model convergence.

5 Urika-XC Quick Reference Information

Log files for a given Urika-XC service are located on the node(s) that the respective service is running on.

- Cray Graph Engine (CGE) - CGE logs are stored in the location specified via the `-l` option of the `cge-launch` command. The default log level of CGE CLI is set to 8 (`INFO`). In addition, the `log-reconfigure` command can also be used to modify log levels. Alternatively, use GUI controls on the **Edit Server Configuration** page to modify log levels. Changing the log level in this manner persists until CGE is shut down. Furthermore, restarting the CGE server is not required if the log level is changed. Restarting CGE reverts the log level to 8 (`INFO`)
- Spark - Default Spark log levels are controlled by the `/tmp/spark/conf/log4j.properties` file. Default Spark settings are used when the system is installed, but can be customized by creating a new `log4j.properties` file. A template for this customization can be found in the `log4j.properties.template` file. The Spark service does not need to be restated if the log level is changed.
    - **Spark event Logs** - Urika-XC stores Spark event logs in per-user directories. By default, the location is `/lus/scratch/sparkHistory/` if it is available, or `$HOME/.minerva/sparkHistory` if it is not. User may override this and select their own event log directory by setting the environment variable `SPARK_EVENT_DIR` prior to running `start_analytics`. Users may copy these event logs to their local machines, and locally execute the Spark History Server or any other tools which parse event logs.
    - **Spark worker logs** - These logs reside in the `$HOME/.minerva/sparkHistory` directory.

Default Port Assignments

Table 2. Default Port Assignments for Urika-XC Services

Service	Default Port
CGE `cge-launch` command	3750. See S-3010, "*Cray® Graph Engine Users Guide*" for more information about the `cge-launch` command or see the `cge-launch` man page.
CGE Web UI and SPARQL endpoints	3756

Major Software Versions

Table 3. Urika-XC Software Component Versions

Software Component	Version
CGE	3.1UP00
Apache Spark	2.1.1
Anaconda Python	4.4.0

Software Component	Version
Dask	0.14.3 and later
Dask distributed	1.16.3 and later
Intel BigDL	0.1.1
Analytic Programming Environment	
Python	Anaconda Python 4.4, including Python versions 2.7, 3.5, and 3.6
Java	openjdk version 1.8.0_131
Scala	2.11.8
R	3.4.0
Maven	3.3.9
SBT	0.13.9
ANT	1.9.2