XC™ Series Urika®-XC Analytic Applications Guide

(1.1.UP00)

S-2589 Rev B

Contents

1 About XC™ Series Urika®-XC Analytic Applications Guide (S-2589) Rev A

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

Publication Title	Date	Release
XC™ Series Urika®-XC Analytic Applications Guide (S-2589)	August, 2017	1.0UP00 release
XC™ Series Urika®-XC Analytic Applications Guide (S-2589)	December, 2017	1.1UP00 release
XC™ Series Urika®-XC Analytic Applications Guide (S-2589) Rev A	January, 2018	1.1UP00 release. Th revision contains minor corrections to the usage of the `run_training` command.
XC™ Series Urika®-XC Analytic Applications Guide (S-2589) Rev B	January, 2018	1.1UP00 release. Th revision contains additional information about the supported version of Dask Distributed.

Scope and Audience

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

Record of Revision

New and updated content since the 1.0UP00 release is listed below.

- **New content:** This publication version contains new information related to:
  - using Jupyter NoteBooks
  - executing commands inside a Shifter container using the `run_training` command.
  - setting up SSH tunnels for UIs.
  - setting up SSH between OSA container nodes.
  - visualizing statistics using TensorBoard.
  - running TensorFlow on XC system GPUs.
  - creating new Conda environments with TensorFlow.
  - training Inception-V3 using GRPC distributed TensorFlow.
- **Updated content:**

- Updates to software component versions.
- Updates to the list of Urika-XC features.

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 runs on the Cray XC series systems. It features a comprehensive analytics software stack for capturing and organizing a wide variety of data types from different sources and for executing a variety of analytic jobs on them. In addition, the Urika-XC software stack features components for performing machine and deep learning tasks.

Urika-XC consists of two components, Open Source Analytics (OSA) and Cray Graph Engine (CGE). They may be installed separately or together. OSA is based on images that run inside Shifter containers, while CGE is a user-level binary application.

Urika-XC software can be used with CLE 6.0 UP02 and later CLE releases.

Features and Analytic Components

- **Support for the Multiple Workload Managers** - Urika-XC supports a number of workload managers, including Slurm, Moab Torque and PBS Pro.
- **Support for Jupyter Notebook** - Urika-XC supports Jupyter Notebook with the Jupyter Notebook server, which is a web application that enables creating and sharing documents that contain live code, equations, visualizations, and explanatory text. For more information, visit *http://jupyter.org*
- **Support for GPUs** - Urika-XC enables running TensorFlow on Xeon and Nvidia GPU nodes.
- **Support for accessing DataWarp Files** - Urika-XC enables users to access files in Cray DataWarp, which provides an intermediate layer of high bandwidth, file-based storage to applications running on compute nodes. For more information, refer S-2558, *XC™ Series DataWarp™ User Guide*
- **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. For more information, refer to *Cray® Graph Engine User Guide*
- **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. For more information, visit *https://spark.apache.org*
    - **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. For more information, visit *https://anaconda.org*
    - **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. For more information, visit *http://dask.pydata.org*

- **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. For more information, visit *https://www.intel.com*
- **TensorFlow™ and TensorBoard** - TensorFlow is a software library for dataflow programming across a range of tasks. It is a Math library, which is also used for machine learning applications, such as neural networks. TensorFlow provides a utility called TensorBoard that can display a picture of the computational graph. For more information, visit *https://www.tensorflow.org*

3 About Open Source Analytics (OSA) Images

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

Only OSA images an CGE 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.

3.1 Resource Allocation

Two types of resource allocation are supported on Urika-XC.

Resource Allocation for CGE

Resource allocation for CGE is described in *About the Cray Graph Engine (CGE)* on page 20 and on the CGE man pages

Resource Allocation for OSA

Urika-XC software can be run on the Slurm, Moab Torque and PBS Pro workload managers. Before an analytics cluster can be started, the desired number of nodes needs to be allocated using the workload manager. 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.
- Slurm, *N*-2 worker containers will be launched.
- PBS Pro, *N*-1 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
```

- Example for PBS Pro:

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

3.2 Shifter Usage

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.

3.3 Start an Analytics Cluster and Run OSA Jobs Using 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, BigDL, TensorFlow, TensorBoard and Jupyter Notebook. It can be considered as an entry point to the 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 the Dask workers and Dask Scheduler with.
- Set up SSH tunnels for UIs.
- Set up SSH between OSA container nodes.

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

> **NOTE:** If these environment variables need to be set, they must be set prior to running `start_analytics`. Setting them at a later point will have no effect.

- `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.

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.

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

3.4 Start Up an Analytics Cluster and the Analytic Programming Environment

Prerequisites

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

About this task

Urika-XC enables 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 if needed. For more information, refer to *Start an Analytics Cluster and Run OSA Jobs Using the start_analytics Command* on page 8

3. Allocate the desired number of nodes and execute the `start_analytics` command.

   Example for Slurm:

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

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

3.5 Execute Commands Inside Shifter Containers Using the `run_training` Script

The `run_training` script executes a command, such as a TensorFlow distributed training Python script, etc., inside a Shifter container on each node. After accepting the command, `run_training` sets up the run-time environment, such as for training applications that may have been written to take advantage of the Cray machine learning plugin. By default, `run_training` will pass (to the user-specified command) a comma-delimited list of

the nodes that were allocated by the user through their workload manager (WLM). This comma-delimited list of nodes will be appended to the end of the command-line arguments of the user-specified command.

While using `run_training`:

- The `-e` option of the `run_training` script activates a Conda environment that is visible to the Conda installed inside the image. This Conda environment can be either one of those provided inside the image or one created by the user outside the image.
- If the `-e` option is specified, and the training job involves TensorFlow, then the TensorFlow libraries expected by Python in the environment are assumed to be installed in that environment.

For a full list of options and more information, refer to the `run_training` man page.

3.6 Apache Spark Support

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 2.2.0.

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 start up 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 2.2.0 builds with Scala 2.11.8.

Urika-XCships 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>2.2.0</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

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. If an application is running out of memory or temporary space, try changing this back to `true`.
- **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 12.

3.7 Enable Anaconda Python and the Conda Environment Manager

About this task

Urika-XC OSA images come with the Anaconda Python distribution version 5.0.0, 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. Load the analytics module

   ```
   $ module load analytics
   ```

2. Allocate resources, using workload management specific commands.

   The following example is specific to Slurm.

   ```
   $ salloc -N numberOfResources
   ```

3. Start an analytics cluster.

   ```
   $ start_analytics
   ```

   For more information, refer to the `start_analytics` man page.

   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.

4. 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
   ```

   IMPORTANT: 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. This is particularly useful when the home directory space is limited.

5. Activate the Conda environment.

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

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

3.8 About Dask

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. 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. Dask 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.

CAUTION: Dask Distributed versions 1.20 and latter are not compatible with Urika-XC. If Conda attempts to install these versions in the environment, users may force the earlier version by manually specifying "`distributed=1.19 bokeh=0.12.7`" while creating the Conda environment.

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.

3.8.1 Use Dask to Run Python Programs

About this task

This procedure provides instructions for using Dask to create a Conda environment and then launching an analytics cluster to run a Python program on the cluster.

Procedure

1. Log on to a login node.
2. Load the `analytics` module.

```
$ module load analytics
```

3. Create a Conda environment with Dask, Dask Distributed packages, as well as any other Python packages and versions to use with Dask.

   This can be done in development mode as well.

   CAUTION:

   Dask Distributed versions 1.20 and latter are not compatible with Urika-XC 1.1. To prevent Conda from installing these versions in the environment, users may force the earlier versions of Dask Distributed and Bokeh (which distributed depends on) by manually specifying "`distributed=1.19 bokeh=0.12.7`" when creating the Conda environment.

Alternatively, the incompatibilities in Dask Distributed 1.20 may be worked around by adding "`use-file-locking: false`" to the end of the *`user_home_directory`*`/.dask/config.yaml` file.

```
$ start_analytics -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
```

4. Allocate resources and start an analytics cluster, using the `--dask`/`-k` option to start Dask and the `--dask-env`/`-e` option to specify the Conda environment.

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

5. 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'])
>>>
```

3.9 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/0.3.0/* and review the section '*Getting Started*' for an introduction to BigDL. In addition, the section '*Programming Guide for BigDL*' covers BigDL concepts and APIs for building deep learning applications.

BigDL on

BigDL is built with MKL support and is pre-installed on . The BigDL distribution package is located under `/opt/bigdl-0.3.0/dist` in the software. The version of BigDL used on is 0.3.0.

Use the following environment variables (which are set automatically) 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.

3.9.1 Run Intel BigDL Programs Using `spark-submit` or `spark-shell`

Prerequisites

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

About this task

BigDL uses the Intel MKL library to achieve high performance. The LeNet on MNIST "Hello World" deep learning example trains LeNet-5 on the MNIST data using BigDL. For more information, visit *https://bigdl-project.github.io/0.3.0/* and see the section titled '*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.

As an example, this is how the user would build the LeNet MNIST example.

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. Optional: Set values for environment variables if needed.
   c. Allocate the desired number of nodes in the interactive mode and execute the `start_analytics` script.

      The following example is specific to Slurm:

      ```
      $ salloc -N 34 start_analytics
      ```

      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. For more information, refer to the `start_analytics` man page.
3. Run the LeNet training as a standard Spark program using `spark-submit`

   ```
   $ 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-0.3.0-jar-with-dependencies.jar \
   -f /dir/username/mnist -b 2560 -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`*, i.e., the product of the number of nodes and the number of cores-per-node.

3.9.2 Run Intel BigDL Programs Using PySpark

Prerequisites

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

About this task

This procedure enables users to run PySpark applications on images using Intel® 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. Optional: Set values for environment variables if needed.
   c. Allocate the desired number of nodes in the `interactive` mode and execute the `start_analytics` script.

      ```
      $ salloc -N 34 start_analytics
      ```

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. For more information, refer to the `start_analytics` man page.

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-0.3.0-python-api.zip
```

5. Set the Python path.

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

6. Use the `spark-submit` command to execute the pyspark test.

   In the preceding, `-b`: Specifies the mini-batch size. It is expected that the mini-batch size is a multiple of *`node_number`* * *`core_number`*, i.e., the product of the number of nodes and the number of cores-per-node

```
$ spark-submit --total-executor-cores 640 --conf spark.executor.instances=32 \
--conf spark.executor.cores=20 --py-files ${PYTHON_API_ZIP_PATH},\
./tests/py_files/v0.3.0_py3/textclassifier.py --jars ${BIGDL_JAR} \
--conf spark.executorEnv.PYTHONHASHSEED=123 \
./tests/py_files/v0.3.0_py3/textclassifier.py -b 2560 --max_epoch 3 --model cnn
```

3.9.3 Get Started with Intel BigDL

Intel® 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.

```
$ 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/0.3.0/* and see '*Training LeNet on MNIST - The "hello world" for deep learning*' in the '*Examples*' section under the '*Scala User Guide*'. 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/0.3.0/*. 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"

$ 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/userName/mnist -b $batch_size -r 0.10 \
--checkpoint ./tests/log/model # echo "Exiting function: launchBigDLJob"
}
```

3.9.4 Run Intel BigDL Programs as Local Java or Scala Programs

Prerequisites

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

About this task

Intel® BigDL can be run on a single node as a local Java or Scala program outside of Spark, as described in the following procedure.

Procedure

1. Load the `analytics` module.

   ```
   $ module load analytics
   ```

2. Optional: Set values for environment variables if needed.

3. 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 or outside of Spark.

   ```
   $ 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:/opt/scala-2.11.8/lib/scala-reflect.jar usersMainClassName
     ```

   - Scala:

     ```
     scala -cp fileName.jar usersMainClassName
     ```

   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.

3.9.5 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 from where `spark-submit` is launched.

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.

4 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.

4.1 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.

4.2 Get Started with Using CGE

Prerequisites

This procedure requires CGE to be installed on the system.

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 text 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 text 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, which should be added to `dataset.nt`, 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/text> ?
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/text> ?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://`*`loginNode`*`:3756` to launch web UI, where *`loginNode`* is the name of the login node the front end is launched from.

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

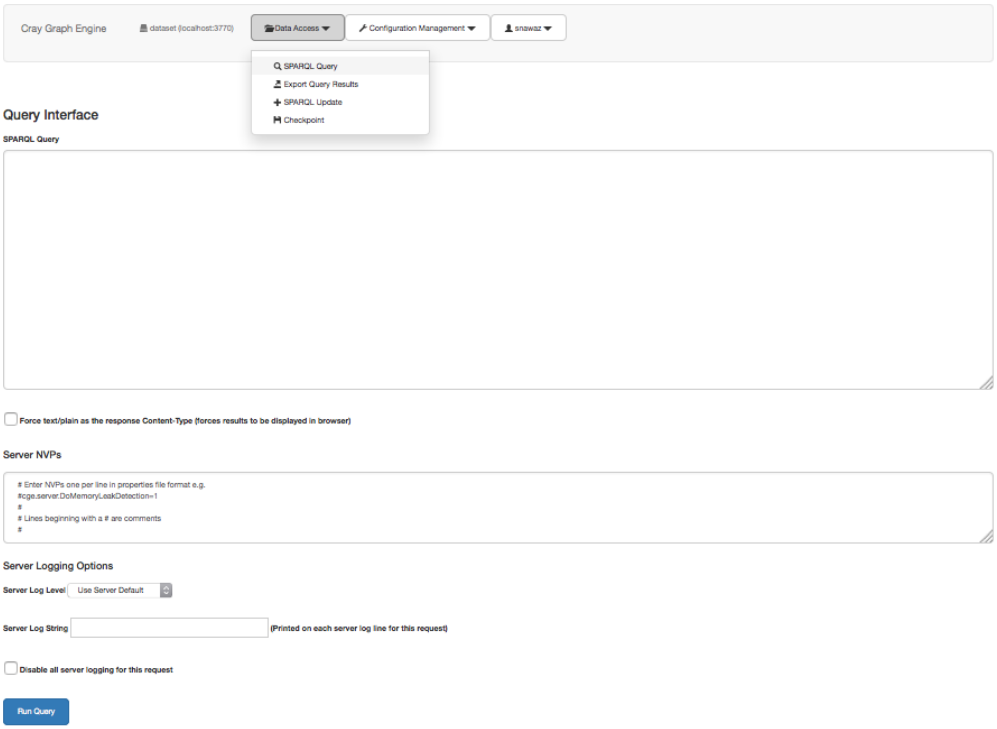
When the CGE front end has been launched, a message similar to the following will be returned on the command-line:

```
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.

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/text> ?
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
```

5 Set Up SSH Tunnels for UIs Using `start_analytics`

About this task

An SSH tunnel can be useful for connecting to a UI running on the interactive node from a different box. One or more SSH tunnels can be set up from the host login node to the interactive node using the `--ssh-tunnel` option of the `start_analytics` command.

In the following instructions:

- *`localPort`* is the user's machine, such as a laptop
- *`loginPort`* is the login node of the XC system
- *`UIport`* is the interactive node

Procedure

1. Log on to a login node.
2. Allocate resources.

   The following example is specific to Slurm.

   ```
   $ salloc -N 4
   ```

3. Load the `analytics` module.

   ```
   $ module load analytics
   ```

4. Start up an analytics cluster with an SSH tunnel from the interactive node to the XC system's login node.

   ```
   $ start_analytics --ssh-tunnel loginPort:UIPort
   ```

   Multiple `--ssh-tunnel` options can be passed to the `start_analytics` command to start up more than one SSH tunnels, as shown in the following example:

   ```
   $ start_analytics --ssh-tunnel loginPort1:UIPort1 --ssh-tunnel loginPort2:UIPort2
   ```

   In the above example, *`UIPort`* and *`loginPort`* are used as examples for ports that the UI under consideration is running on the interactive node, and forwarded to on the login node, respectively. This mechanism can be used to launch TensorBoard and Jupyter Notebook at the same time.

6 Execute a Simple Jupyter NoteBook

About this task

This procedure provides instructions for executing Jupyter Notebooks on the system.

Procedure

1. Log on to a login node.
2. Obtain a job allocation.

   The following example is specific to Slurm:

   ```
   $ salloc -N 4
   salloc: Granted job allocation 7983
   salloc: Waiting for resource configuration
   salloc: Nodes nid00[180-183] are ready for job
   ```

3. Load the `analytics` module

   ```
   $ module load analytics
   ```

4. Execute the `start_analytics` command, specifying the login and UI ports.

   Running with the `--login-port` and `--ui-port` options also automatically sets the `JUPYTER_RUNTIME_DIR` environment variable. If this variable is not set to a writeable directory, Jupyter will not run inside containers.

   ```
   $ start_analytics --login-port loginPort --ui-port UIPort
   ```

   Here

   - *`loginPort`* is the port to use on the login node.
   - *`UIPort`* is the port that the UI runs on.

   Alternatively, perform both the preceding steps in one go, as shown in the following example:

   ```
   $ salloc -N 4 start_analytics --login-port loginPort --ui-port UIPort
   ```

5. Start the Jupyter Notebook application.

   To use Jupyter with a Conda environment, install Jupyter in the Conda environment, and activate the environment before running the `jupyter notebook` command.

   The following example assumes that Jupyter Notebook is not being used with a Conda environment.

   ```
   $ jupyter notebook --port UIPort
   [I 20:23:57.376 NotebookApp] Serving notebooks from local directory: /home/
   ```

```
users/username
[I 20:23:57.376 NotebookApp] 0 active kernels
[I 20:23:57.376 NotebookApp] The Jupyter Notebook is running at: http://
localhost:9100/?token=6aacf7f9e13c412921a4fde10ae51d638065f60839114193
[I 20:23:57.376 NotebookApp] Use Control-C to stop this server and shut down
all kernels (twice to skip confirmation).
[W 20:23:57.380 NotebookApp] No web browser found: could not locate runnable
browser.
[C 20:23:57.381 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:9100/?
token=6aacf7f9e13c412921a4fde10ae51d638065f60839114193
```

6. Create an SSH tunnel from the localhost to the XC login node in a new terminal window.

   ```
   $ ssh -L localPort:localhost:loginPort hostname
   ```

   Here, *loginPort* should match the login port specified in step 4. *localPort* is the port to use to view the UI from on the local machine. *hostname* is the login node that `start_analytics` was run on in step 4.

7. Copy and paste this URL into a browser when connecting for the first time.

   To login with a token, point a browser at `http://localhost:`*localPort*`/?`. Enter the received token when prompted.

8. Shut down the Jupyter Notebook server by killing the Jupyter process on the interactive node.

7 Visualize Statistics with TensorBoard

About this task

TensorBoard is a set of web applications that can be used for analyzing TensorFlow graphs. This procedure helps getting starting with using TensorBoard.

For more information, visit *https://www.tensorflow.org*.

Procedure

1. Allocate resources.

   ```
   $ salloc -N numberOfNodes
   ```

2. Load the analytics module.

   ```
   $ module load analytics
   ```

3. Start an analytics cluster using one of the following mechanisms.

   - ```
     $ start_analytics --login-port loginPort --ui-port UIPort
     ```
   - ```
     $ start_analytics --ssh-tunnel loginPort:UIPort
     ```

     This mechanism will automatically tunnel the UI port of the interactive node to `loginPort` on the login node.

4. Run the TensorFlow or BigDL application with instrumented code to generate TensorBoard summary data and store the summary data in a directory of choice.

   In this procedure it is assumed that the summary data is stored in *`logDirName`*.

5. Run TensorBoard after activating a sample TensorFlow Conda environment.

   ```
   $ tensorboard --logdir="logDirName" --port=UIPort
   ```

   TensorBoard can be started even when the application is running. The statistics can be visualized as the training progresses. Another approach is to run TensorBoard after the training to perform post-run analysis.

6. Create a tunnel to the login node port on the login node.

   ```
   $ ssh -L localPort:localhost:loginPort hostname
   ```

   Here, `loginPort` should match the login port specified in step 3. `localport` is the port it is required to view TensorBoard the UI from on the user's machine. *`hostname`* is the login node that `start_analytics` was run on in step 3.

For example, if 7801 is specified as the `loginPort` and it is required to view TensorBoard on the local machine on port 7800, execute:

```
$ ssh -L 7800:localhost:7801 hostname
```

7. Point a browser at `localhost:`*`localPort`* to visualize TensorBoard.

   For example, if the local port is 7800, point a browser at `localhost:7800`

   If multiple users are running TensorBoard, ensure that the ports being used are unique. For example, in addition to the above run of TensorBoard, another user may be running another TensorFlow or BigDL application and may want to run TensorBoard. In such cases, it is important to ensure that the *`UIPort`* is forwarded to the host on interactive node.

   This can be achieved by performing the following tasks:

   1. Add additional ports to `start_analytics`

      Pass a unique login port to `start_analytics`. For example, if the login port 7801 is busy, pass this login port to `start_analytics` as follows:

      ```
      $ start_analytics --login-port 7802 --ui-port 7800
      ```

      To check if a port is in use, execute:

      ```
      $ nc -z localhost PORT_NUMBER
      $ echo $?
      ```

      The port specified is available for use if the preceding command returns 1.

   2. Run TensorBoard.

      ```
      $ tensorboard --logdir="logDirName" --port=UIPort
      ```

   3. Open another terminal window on the local machine and execute:

      ```
      $ ssh -L localPort:localhost:loginPort hostName
      ```

      For example, if the local port is 7800 and login port is 7802, run:

      ```
      $ ssh -L 7800:localhost:7802 hostname
      ```

   4. Open TensorBoard, by pointing a local browser at `localhost:7800` to visualize statistics from the second application.

   For more information, refer to the `start_analytics` man page.

8 Set up SSH Between OSA Container Nodes

Prerequisites

This procedure requires the Shifter configuration to use the Shifter SPANK plugin for Slurm. For more information, refer to *https://github.com/NERSC/shifter/wiki/SLURM-Integration*.

About this task

This procedure can be used to SSH between the Open Source Analytics (OSA) container nodes. It is currently only supported on systems that use Slurm as their workload manager.

Procedure

1. Log on to a login node.
2. Load the analytics module.

   ```
   $ module load analytics
   ```

3. Allocate the desired number of nodes, specifying the image.

   ```
   $ salloc -N 10 --image=$ANALYTICS_IMG
   ```

   Here `$ANALYTICS_IMG` is the environment variable that specifies the image to load into the container. This variable is set automatically when the user executes the `module load analytics` command.

   This command will return a list of node IDs of the allocated nodes.

4. Start the analytics cluster, specifying the `-s`/`-ssh` option.

   ```
   $ start_analytics -s
   ```

5. Verify that it is possible to SSH between the cluster nodes by attempting to SSH to a node, using one of the node IDs returned in step 1.

9 Run TensorFlow with the Cray PE Machine Learning Plugin

Prerequisites

This procedure requires:

- Urika-XC software with Cray programming environment machine learning plugin for using `run_training` examples.
- The CuDNN library is required for running TensorFlow on GPU nodes. Users may need to download CuDNN from NVIDIA if their site does not already have it installed.

About this task

The Cray Programming Environment Machine Learning plugin (CPE ML plugin) enables scaling and significantly higher productivity to deep learning (DL) frameworks. This capability is intended for users needing faster time to accuracy and is based on data-parallel DL training. TensorFlow users on Urika-XC start with a serial (non-distributed) Python training script, include a few simple lines for the CPE ML Plugin, and are then able to train across many nodes at very high performance. User that already have distributed gRPC-based Python training script can also use the CPE ML plugin to obtain better performance by by-passing gRPC setup. The CPE ML plugin has both C and Python interfaces for the communication needs of DL training.

Modifying a TensorFlow Training Script to use the CPE ML Plugin

The CPE ML plugin module includes two examples of training scripts modified to use the plugin. The modifications needed include:

- A call to the initialize the CPE ML plugin
- A call to broadcast initial model parameters to all ranks
- Possible modifications to learning rate decay schedules and other mini-batch size dependent parameters to account for the effective mini-batch size across all processes
- A call to communicate gradients among processes after local gradient calculation but before applying gradients
- A call to finalize the CPE ML plugin

About `MNIST` and `tf_cnn_benchmarks`

- **`MNIST`**- This is an example of modifying a serial training script to use the CPE ML plugin. The script is available in `/opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_mnist/mnist.py`. The script is documented with any modifications, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_mnist/README` also describes the modifications.

- **`tf_cnn_benchmarks`** - This is an example of modifying a script already able to run across multiple nodes through gRPC to instead use the CPE ML plugin. Both capabilities (gRC and the Plugin) are available as options in this script, and the script can be used to benchmark scaling and performance of various CNNs using either gRPC or CPE ML Plugin. The source files for this benchmark are located in: `/opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_cnn_benchmarks`. Any modifications are documented inside the source files, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_cnn_benchmarks/README` describes the changes in detail.

About the `run_training` script

The `run_training` script allows the user to execute a distributed job using MPI or the Cray programming environment machine learning plugin. The user specifies the number of processes to run on each allocated node via the `-ppn` argument, and also specifies how many processes to run across all allocated nodes via the `-n` argument, as shown in this procedure.

Procedure

1. Load the `analytics` module.

   ```
   $ module load analytics
   ```

2. Allocate the desired number of nodes in interactive mode or as part of a SLURM or PBS job submission script. If the XC system being used has GPUs, and it is required to use them for TensorFlow, be sure to add options for requesting nodes with GPUs.

   An example of SLURM using an interactive session requesting two NVIDIA P100 nodes is shown below (users should refer to documentation provided by their site for exact allocation syntax):

   ```
   $ salloc --nodes=2 --exclusive --gres=gpu -C P100
   ```

   For PBS, a similar request may look like:

   ```
   $ $ qsub -I -l nodelist=GPUNodeIDs -l nodes=2
   ```

3. Switch to the current working directory to copy the contents of `/opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_cnn_benchmarks/*` (which are the TensorFlow examples packaged with the plug-in) to the current working directory if it is required to run the `tf_cnn_benchmark` example provided with the CPE ML plug-in.

   ```
   $ cd workingDir
   $ cp -r /opt/cray/pe/craype-ml-plugin-py3/1.0.1/examples/tf_cnn_benchmarks/* .
   ```

4. Execute the training script with `run_training`.

5. Submit a TensorFlow command to the `run_training` script.

   If the Cray PE machine learning plugin is installed on the system, it can be used as a test case in this step. This procedure assumes the plugin is installed.

   GPU example using 2 nodes with one process per node with user's CuDNN v5.1 library located at `/home/users/alice/CuDNN/cudnn-8.0-v51/cuda/lib64`

   ```
   $ run_training -n 2 --ppn 1 --cudnn-libs /home/users/alice/CuDNN/cudnn-8.0-v51/cuda/lib64 \
   --no-node-list "python tf_cnn_benchmarks.py --num_gpus=1 --batch_size=64 --model=inception3 \
   ```

```
--train_dir=/home/users/alice/tf_cnn_train --data_name=imagenet --variable_update=ps_ml_comm \
--num_intra_threads=1 --local_parameter_device=gpu"
```

CPU without the need for CuDNN `v5.1.num_intra_threads` should be set to the number of cores available on the Xeon or Xeon Phi node. On Xeon Phi users should set `num_inter_threads` to 2 to use additional hyper threads. Users can obtain cudnn libraries from *https://developer.nvidia.com/cudnn*.

Intel Xeon example for Broadwell dual socket 18 core nodes:

```
$ run_training -n 2 --ppn 1 --no-node-list "python tf_cnn_benchmarks.py \
--device=cpu --num_intra_threads=36 --mkl=True --batch_size=64 \
--train_dir=/home/users/alice/tf_cnn_train --data_name=imagenet --variable_update=ps_ml_comm \
--data_format=NHWC --local_parameter_device=cpu"
```

Intel Xeon Phi example for KNL single socket 64 core nodes:

```
$ run_training -n 2 --ppn 1 --no-node-list "python tf_cnn_benchmarks.py \
--device=cpu --num_intra_threads=64 --num_inter_threads=2 --mkl=True --batch_size=64 \
--train_dir=/home/users/alice/tf_cnn_train --data_name=imagenet --variable_update=ps_ml_comm \
--data_format=NHWC --local_parameter_device=cpu"
```

To use the CuDNN library inside containers interactively via the `start_analytics` command, specify the CuDNN libraries via the `--cudnn-libs` option, as shown in the following example:

```
$ start_analytics --cudnn-libs /home/users/username/CuDNN/cudnn-8.0-v51/cuda/lib64
```

For more information, refer to the `start_analytics` and `run_training` man pages.

Additional Help and Tuning Options

To access more information about using and tuning the CPE plugin users can load the following module:

```
$ module load craype-ml-plugin-py3
```

The `intro_ml_plugin` describes the C interface and environment variables for tuning performance. The Python interface is documented in the Python module. To view this information after load the module

```
$ python
>>> import ml_comm as mc
>>> help(mc)
```

10 Create New Conda Environments with TensorFlow

About this task

By default two TensorFlow libraries of versions 1.3 built for Python 3.6 are installed in `/opt/tensorflow_cpu` and `/opt/tensorflow_gpu`. One version is for systems that use only CPUs, whereas the other can be used on systems that have a combination of CPUs and GPUs.

The Urika-XC image contains two sample Conda environments with TensorFlow for Python 3.6:

- `py36_tf_cpu` for systems using CPUs only
- `py36_tf_gpu` for systems using both CPUs and GPUs

Users can activate these environments according to their platforms.

The `run_training` script has an option to automatically activate a Conda environment via the `-e` option. The wheels for these TensorFlow builds are available inside the image. There are 2 additional wheels provided for TensorFlow built for Python 2.7, one for systems using CPUs only and one for systems using both CPUs and GPUs.

The locations of these four wheels are:

- Versions for CPUs only: `/opt/tensorflow_cpu-1.3.0/wheel`
- Version for CPUs and GPUs: `/opt/tensorflow_gpu-1.3.0/wheel`

To run Python 2.7 TensorFlow inside the image, the user can create a new Python 2.7 Conda environment along with `pip` and install one of the wheels provided in the image. The user can also activate their own environment by specifying it via the `-e` option to the `run_training` script.

The following items should be kept under consideration while using the `-e` option:

- The `-e` option of the `run_training` script activates a Conda environment that is visible to the Conda installed inside the image. This Conda environment can be either one of those provided inside the image or one created by the user outside the image.
- If `-e` option is specified, and the training job involves TensorFlow, then the TensorFlow expected by the Python in the environment is assumed to be installed in that environment.

Use the following instructions to create a new environment with TensorFlow for Python 2.7 for CPUs.

Procedure

1. Log on to a login node.
2. Load the `analytics` module

```
$ module load analytics
```

3. Obtain a Slurm job allocation.

```
$ salloc -N 4 start_analytics
salloc: Granted job allocation 7983
salloc: Waiting for resource configuration
salloc: Nodes nid00[180-183] are ready for job
```

4. Execute the following in the analytics shell.

```
$ conda create -n python2 python=2.7 pip
$ source activate python2
$ pip install /opt/tensorflow_cpu-1.3.0/wheel/tensorflow-1.3.0-cp27-cp27mu-manylinux1_x86_64.whl
```

5. Exit the cluster.

```
$ exit
```

6. Execute commands as needed in the new environment.

```
$ run_training -e python2 command
```

11 Train Inception-V3 Using GRPC-Distributed TensorFlow

About this task

The GRPC protocol provides features such as authentication, bidirectional streaming, flow control, blocking/nonblocking bindings, cancellation and timeouts. It generates cross-platform client and server bindings for many languages.

The `run_training` command can be used to train distributed TensorFlow applications with the GPRC protocol on CPUs and GPUs. It can also be used to train distributed TensorFlow applications with the GPRC protocol within a Conda environment.

To learn more about TensorFlow and Inception-V3, visit *https://www.tensorflow.org*.

Cray recommends using the Cray PE machine learning plugin for optimal scaling of distributed TensorFlow. However, Urika-XC also provides the option to use GRPC based distributed TensorFlow instead of the machine learning plugin.

Procedure

1. Log on to a login node.
2. Load the `analytics` module.

   ```
   $ module load analytics
   ```

3. Clone the contents of the `inception` directory of the TensorFlow models.

   ```
   # git clone https://github.com/tensorflow/models
   ```

4. Prepare the data in TensorFlow format.
   a. Download and convert the ImageNet data to native TFRecord format.

      To learn more about ImageNet, visit *http://www.image-net.org*. The TFRecord format consists of a set of sharded files, where each entry is a serialized `tf.Example` proto. Each `tf.Example` proto contains the `ImageNet` image (JPEG encoded) as well as metadata, such as label and bounding box information.

   b. Sign up for an account with ImageNet to gain access to the data by locating the sign up page, creating an account and requesting an access key to download the data.

   c. Specify the location where the ImageNet data should be placed.

      ```
      DATA_DIR=$HOME/imagenet-data
      ```

   d. Build the preprocessing script.

```
$ cd inception
$ bazel build //inception:download_and_preprocess_imagenet
```

e. Execute the preprocessing script.

```
$ bazel-bin/inception/download_and_preprocess_imagenet "${DATA_DIR}"
```

f. Enter the username and password when prompted. Enter the username and password when prompted. Once these values are entered, the script doe not require any further user interaction.

The final line of the output script should contain the following:

```
Finished writing all 1281167 images in data set.
```

When the script finishes executing, `DATA_DIR` will contain 1024 training files and 128 validation files. The files will match the patterns `train-?????-of-01024` and `validation-?????-of-00128`, respectively.

5. Copy over the `tensorflow-examples/inception/imagenet_distributed_train.py.slurm` and `tensorflow-examples/inception/inception_distributed_train.py` file (which are the Inception and Imagenet distributed train files) to the `inception/inception/` directory.

```
$ cp tensorflow-examples/inception/imagenet_distributed_train.py.slurm inception/inception/
$ cp tensorflow-examples/inception/inception_distributed_train.py inception/inception/
```

6. Edit the `tensorflow_dist_inception.example` template located under `tensorflow-examples/inception` to specify values for the required variables.

- `INCEPTION_DIR`: The location of inception model. This is the Inception directory under the TensorFlow models directory, which was cloned above.
- `IMAGENET_DIR`: The location of `imagenet_distributed_train.py.slurm`.
- `IMAGENET_TRAIN_DIR`: The location for training results and the model checkpoint.
- `IMAGENET_DATA_DIR`: The location of the `imagenet` data.

7. Run the training using the `run_training` command.

`run_training` takes one mandatory argument, namely a command *`CMD`* (e.g., a TensorFlow distributed training Python script) to run inside the Shifter container on each node. Once provided with this mandatory argument (and possibly optional arguments), `run_training` sets up the run-time environment, e.g., for training applications that may have been written to take advantage of the Cray ML plugin. By default `run_training` will pass to *`CMD`* a list of comma-delimited list of nodes, previously allocated by the user through their work load manager (WLM). The CMD is responsible for using these nodes, such as, for distributed training. In case the user application does not expect or may fail upon receiving extra arguments, the passing of node list may be suppressed by providing the command-line option `--no-node-list` to `run_training`.

To execute `run_training` on CPUs, execute:

```
$ run_training tensorflow-examples/inception/tensorflow_dist_inception.example
```

To train a model using GPU, download the cudNN libraries from *https://developer.nvidia.com/cudnn* into a directory which is provided to the `run_training` command using the `--cudnn-libs` option, and then specify the path to the `cudNN` libraries as follows:

In the following examples, */path/to/cudnn* is used as an example to the path to cudNN libraries.

```
$ run_training tensorflow-examples/inception/tensorflow_dist_inception.example \
--cudnn-libs /path/to/cudnn/cudnn-8.0-v51/cuda/lib64
```

The user can also specify a Conda environment to execute TensorFlow applications in. For example, to use a different version of the `numpy` library that is installed in a Conda environment named `e`, execute:

```
$ run_training --env e --cudnn-libs /path/to/cudnn/cudnn-8.0-v51/cuda/lib64 \
tensorflow-examples/inception/tensorflow_dist_inception.example
```

This will activate the Conda environment named `e` before the application is run.

12 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 on the local nodes they run on.

DataWarp Access from Shifter Containers

To access DataWarp from Shifter containers, an admin would need to edit the `/etc/opt/cray/shifter/udiRoot.conf` file's `siteFs` parameter to add the following:

```
/var/opt/cray/dws:/var/opt/cray/dws:rec:slave
```

For more information, refer to S-2571, '*XC™ Series Shifter User Guide*'.

Default Port Assignments

Table 1. 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 2. Urika-XC Software Component Versions

Software Component	Version
CGE	3.2UP00
Apache Spark	2.2.0
Anaconda Distribution of Python	5.0.0
Dask	0.14.3 and later
Dask Distributed	1.16.3 to 1.19.1
Intel BigDL	0.3.0
Analytics Programming Environment	
Python	3.6 as part of Anaconda 5.0.0. Anaconda also supports creating `pythong` environments with 2.7, 3.4, and 3.5
Java	1.8
Scala	2.11.8
R	3.4.1
Maven	3.3.9
SBT	0.13.9
ANT	1.9.2
TensorFlow	1.3.0
Jupyter NoteBook	4.3.0