XC™ Series Urika®-XC Analytic Applications Guide

(1.2.UP00)

S-2589

Contents

1 About XC™ Series Urika®-XC Analytic Applications Guide (S-2589) 4
2 About Urika-XC 6
2.1 About Open Source Analytics (OSA) Images 7
2.2 Resource Allocation 8
2.3 Shifter Usage 8
3 Perform Machine and Deep Learning Tasks 9
3.1 Start an Analytics Cluster and Run OSA Jobs Using the `start_analytics` Command 9
3.2 Execute Commands Inside Containers Using the `run_training` Script 10
3.3 Apache Spark Support 10
3.4 Use Dask to Run Python Programs 12
3.5 Get Started with Intel BigDL 13
3.5.1 Run Intel BigDL Programs Using PySpark 15
3.5.2 Run Intel BigDL Programs as Local Java or Scala Programs 16
3.5.3 Run Intel BigDL Programs Using `spark-submit` or `spark-shell` 17
3.6 Get Started with PyTorch 18
3.6.1 Run PyTorch on a Single Node Inside the Container Using the `start_analytics` Command 19
3.6.2 Run an MPI Application Using the run_training Command 20
3.7 Enable Anaconda Python and the Conda Environment Manager 21
3.8 Create New Conda Environments with TensorFlow 22
3.9 Train Inception-V3 Using gRPC Distributed TensorFlow 23
3.10 Use `run_pbdR` to Execute an R MPI Application Inside an Image 24
3.11 Run TensorFlow with the Cray Programming Environment Machine Learning Plugin 26
3.12 Use PyTorch with the Programming Environment (PE) Plugin 28
3.13 Use Keras with the Cray Programming Environment (PE) Plugin 30
3.14 Run Keras Using the Tensorflow Backend 32
3.15 Run Horovod with Tensorflow, Keras, or PyTorch 33
3.16 Select GNU Version 34
3.17 Execute a Simple Jupyter NoteBook 34
3.17.1 Run R in a Jupyter Notebook with the IRKernel Package 36
3.18 Visualize Statistics with TensorBoard 36
4 Set up Connectivity 39
4.1 Set up Connectivity to User Interfaces 39
4.2 Set up Connectivity Between OSA Container Nodes 40
5 About the Cray Graph Engine (CGE) 42

5.1 CGE Features....42
5.2 Get Started with Using CGE....42
6 Hyperparameter Optimization (HPO) Support....47
6.1 Get Started with Using Hyperparameter Optimization (HPO)....48
6.2 Hyperparameter Optimization (HPO) Examples....50
6.3 Hyperparameter Optimization (HPO) Troubleshooting Information....55
7 Urika-XC Quick Reference Information....57
7.1 BigDL Logging....60

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

The *XC™ Series 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.

Publication Title	Date	Release
XC™ Series Urika®-XC Analytic Applications Guide (S-2589)	November, 2018	Urika-XC 1.2UP00
XC™ Series Urika®-XC Analytic Applications Guide (S-2589) Rev B	January, 2018	Urika-XC 1.1UP00
XC™ Series Urika®-XC Analytic Applications Guide (S-2589) Rev A	January, 2018	Urika-XC 1.1UP00
XC™ Series Urika®-XC Analytic Applications Guide (S-2589)	December, 2017	Urika-XC 1.1UP00
XC™ Series Urika®-XC Analytic Applications Guide (S-2589)	August, 2017	Urika-XC 1.0UP00

Scope and Audience

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

Record of Revision

New and updated content since the Urika-XC 1.2UP00 release is listed below.

- **New content:**
  - Procedure to *Run R in a Jupyter Notebook with the IRKernel Package* on page 36
  - Information about pbdR and procedure to *Use run_pbdR to Execute an R MPI Application Inside an Image* on page 24
  - Information about Horovod and procedure to *Run Horovod with Tensorflow, Keras, or PyTorch* on page 33.
  - Information about PyTorch and procedures to:
    - *Get Started with PyTorch* on page 18.
    - *Run PyTorch on a Single Node Inside the Container Using the start_analytics Command* on page 19.
    - *Execute R MPI Applications Inside an Image Using the run_pbdR Command*.

  - Information about Keras and procedures to:
    - *Use PyTorch with the Programming Environment (PE) Plugin* on page 28
    - *Run Keras Using the Tensorflow Backend* on page 32
    - *Use Keras with the Cray Programming Environment (PE) Plugin* on page 30
- **Updated content:**
  - *Urika-XC Quick Reference Information*
  - *About 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 software stack, which is optimized for performing machine learning and AI related tasks. It runs on the Cray XC series systems.

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

Urika-XC software can be used with CLE 6.0 UP05 and later CLE releases.

CAUTION: Urika-XC does not support Shifter versions that are not part of CLE 6.0 UP05 and later CLE releases.

Features and Analytic Components

- **Support for 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.
- **Support for GPUs** - Urika-XC enables running TensorFlow on Xeon CPU nodes 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 to 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*.
- **Cray Programming Environment Machine Learning plugin** - The Cray Programming Environment Machine Learning plugin enables significantly increasing productivity of deep learning (DL) frameworks. This capability is intended for users needing faster time to accuracy and is based on data-parallel DL training. The CPE ML plugin has both C and Python interfaces for the communication needs of DL training.
- **HPO** - The `crayai hpo` Python library enables users to optimize machine learning and deep learning models by traversing possible hyperparamter combinations. A hyperparamter is a value that defines either network structure, such as number of convolutional layers, or the training process, such as learning rate. The `crayai hpo` library provides three underlying algorithms for HPO: grid, random and genetic. It can also be used for Population-Based Training (PBT) via the genetic algorithm.
- **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** - Anaconda is a distribution of the Python 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*
- **R** - R is both a programming language and an integrated environment for statistical computing and graphics. Urika-XC provides an optimized version of R built using OpenBLAS for performance.
- **PyTorch** - PyTorch is an open source optimized tensor library and deep learning framework for Python. Although it is based on Torch (a Lua based deep learning framework), PyTorch is built to be deeply integrated into Python.
- **Programming with Big Data in R (pbdR)** - pbdR is a set of highly scalable R packages for distributed computing and statistics. The pbdR ecosystem also includes advanced profiling tools, large scale I/O infrastructure, and client/server frameworks.
- **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.
- **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 displays a picture of the computational graph.
- **Keras** – Keras is high-level API and an open source Python library that runs on top of other deep learning frameworks. Keras enables defining and training neural network models in a few short lines of code, making it an attractive option for quick prototyping. The default backend framework is TensorFlow; CNTK is also supported.
- **Horovod** - Horovod is a distributed training framework, through which deep learning models can be trained across Cray XC systems. Horovod is built to use Cray MPI on XC systems. Urika-XC features a version of Horovod that has been optimized for CPU and GPU support.

Please refer to online documentation for detailed descriptions of each of these software components.

2.1 About Open Source Analytics (OSA) Images

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

Only OSA images and 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.

2.2 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 42 and 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 system's 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/PBS Pro:

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

2.3 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 Urika-XC. Shifter also provides per-node cache functionality that creates a loop back mounted file system on every node. This provides efficient emulation of local storage for frameworks that require it, such as Spark.

3 Perform Machine and Deep Learning Tasks

3.1 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, PyTorch, Keras, TensorFlow, TensorBoard and Jupyter Notebook. It can be considered as an entry point to the OSA components. The `start_analytics` command normally starts an analytics cluster within the nodes of a user's job allocation.

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.

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 is it required to set these environment variables, 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. In some environments, this may allow better external connectivity for build and environment tools that need to download new packages.
- `SPARK_EVENT_DIR` - Sets the location for Spark event logs.

The `start_analytics` script also features the `-d` option that starts a single analytics container on the current login node. No job allocation is required in this case and 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 may provide better access to the external network in some environments, it can be useful for downloading new packages for builds.

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.

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

3.2 Execute Commands Inside Containers Using the `run_training` Script

The `run_training` script executes commands inside a Shifter container on each node. After receiving 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 ML PE 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.

3.3 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 2.3.2.

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

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" % "2.3.2"
```

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>2.3.2</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*.

Conda Environments

When the system is running in the default mode, 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 21.

When the system is running in the secure mode, Spark jobs are not aware of Conda environments or user Python versions.

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 network on the system 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.

3.4 Use Dask to Run Python Programs

About this task

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.

For more information, refer to *Use Dask to Run Python Programs* on page 12

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.

This procedure provides instructions for creating a Conda environment and running Dask in that environment.

Procedure

1. Log on to a login node.
2. Skip this step if the system does not recognize modules/module files and the Urika®-XC `bin` directory is included in the `PATH`. If the system does use modules, 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 the development mode as well.

CAUTION:

Dask Distributed version 1.20 is not compatible with Urika®-XC. If Conda attempts to install this version in the environment, users may force the earlier version by manually specifying "`distributed=1.19 bokeh=0.12.7`" while 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. This issue has been resolved with Dask Distributed 1.21

```
bash-4.2$ conda create --name mydaskenv dask distributed = 1.19 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.

   Example for Slurm

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

   Example for PBS Pro

   ```
   $ qsub -I -lnodes=numberofNodes
   $ module load analytics
   $ 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'])
   >>>
   ```

3.5 Get Started with 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 Gaffe or Torch models into Spark programs using BigDL.

For more information, visit *https://bigdl-project.github.io/0.7.0/* 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 Urika-XC

The version of BigDL used on is 0.7.0. BigDL is built with MKL support and is pre-installed on . The BigDL distribution package is located under `/opt/bigdl-0.7.0/dist` in the software.

Use the following environment variables (which are set automatically) to perform deep learning tasks with the BigDL toolkit:

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

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/0.7.0/*. 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.resnet.TrainCIFAR10 \
$BIGDL_DIR/lib/bigdl-0.7.0-jar-with-dependencies.jar \
```

```
-f /lus/snx11254/userName/mnist -b $batch_size -r 0.10 \
--checkpoint ./tests/log/model # echo "Exiting function: launchBigDLJob"
}
```

3.5.1 Run Intel BigDL Programs Using PySpark

Prerequisites

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

      Example for Slurm:

      ```
      $ salloc -N numberofNodes start_analytics
      ```

      Example for PBS Pro:

      ```
      $ qsub -I -l nodes=numberofNodes
      $ module load analytics
      $ 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. Create a variable for Python libraries.

```
$ export PYTHON_API_ZIP_PATH=${BIGDL_DIR}/lib/bigdl-0.7.0-python-api.zip
```

4. Set the Python path.

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

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

   In the following commands, `-b` option 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/v0.7.0_py3/textclassifier.py --jars ${BIGDL_JAR} \
--conf spark.executorEnv.PYTHONHASHSEED=123 \
./tests/py_files/v0.7.0_py3/textclassifier.py -b 2560 --max_epoch 3 --model cnn
```

3.5.2 Run Intel BigDL Programs as Local Java or Scala Programs

Prerequisites

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

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. Start the analytics cluster.

```
$ start_analytics -d
```

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

4. 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.

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

Prerequisites

This procedure assumes that the workload manager being used is either Moab Torque, Slurm 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/0.7.0/* 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.

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/0.7.0/* 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 numberofNodes start_analytics
```

The following example is specific to PBS Pro:

```
$ qsub -I -l nodes=numberofNodes
$ module load analytics
$ 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-0.7.0-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.

3.6 Get Started with PyTorch

PyTorch is an open source optimized tensor library and deep learning framework for Python. Although it is based on Torch (a Lua based deep learning framework), PyTorch is designed to be deeply integrated into Python.

PyTorch can be run inside the Urika-XC image on a single compute node using Python multi-processing via the `start_analytics` command.

The PyTorch library consists of the following components:

Table 1. PyTorch Library Components

Component	Description
`torch`	Tensor library that provides both CPU and GPU support.
`torch.autograd`	Tape-based automatic differentiation library that supports all differentiable Tensor operations in Torch.
`torch.nn`	Neural networks library.
`torch.multiprocessing`	Used for Python multiprocessing.
`torch.utils`	DataLoader, Trainer and other utility functions.

Component	Description
`torch.legacy(.nn/.optim)`	Legacy code that has been ported over from Torch for backward compatibility reasons.
`torch.distributed`	Provides an MPI-like interface for exchanging tensor data across multiple nodes. It supports a few different backends and initialization methods.

3.6.1 Run PyTorch on a Single Node Inside the Container Using the `start_analytics` Command

Prerequisites

This procedure assumes that Slurm is being used as the workload manager.

About this task

This procedure shows how to use the Urika-XC `start_analytics` command to run the PyTorch MNIST example.

Procedure

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

   ```
   $ module load analytics
   ```

3. Download PyTorch examples from *https://github.com/pytorch/examples.git*

   ```
   $ git clone https://github.com/pytorch/examples.git
   ```

4. Switch to the `examples/mnist` directory.

   ```
   $ cd examples/mnist
   ```

5. Allocate a node in the interactive mode and execute the `start_analytics` command.

   Example for Slurm:

   ```
   $ salloc -N 1 start_analytics
   ```

6. Set the `PYTHONPATH` depending on the type of the node.

   There are separate GPU and CPU builds available on the image at locations `/opt/pytorch_gpu` and `/opt/pytorch_cpu` respectively. If `start_analytics` is used to launch the Urika-XC image, set `PYTHONPATH` to point to the corresponding locations. On the other hand, if `run_training` is used, the runtime scripts in Urika-XC image will automatically pick the right version.

   The following example is for setting `PYTHONPATH` for CPU nodes.

```
$ export  PYTHONPATH=/opt/pytorch_cpu:$PYTHONPATH
```

7. From the bash shell run the PyTorch MNIST example.

```
$ python main.py
```

3.6.2 Run an MPI Application Using the run_training Command

Prerequisites

This procedure assumes that Slurm is being used as the workload manager.

About this task

For distributed computing, PyTorch can be run using the `torch.distributed` package via the `run_training` command using MPI. The Urika-XC image contains the `torch.distributed` package built with Cray MPI support.

Following is a simple MPI application that uses Torch distributed with MPI as the backend:

```
"""simple_mpi.py:"""
#!/usr/bin/env python

import os
import torch
import torch.distributed as dist
from torch.multiprocessing import Process


# Examples from:
# https://github.com/pytorch/tutorials/blob/master/intermediate_source/\
# dist_tuto.rst

"""Do broadcast from rank 0."""
def run(rank, size):
    print('Hello from rank', rank, 'of world size', size)
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 7
    dist.broadcast(tensor, 0)
    if rank == (size-1):
        print('Rank', rank, 'received tensor:', tensor)


# For MPI backend the rank/size will be determined by the MPI rt so
# the params don't have any use here.
#def init_processes(rank, size, fn, backend='tcp'):
def init_processes(fn, backend='mpi'):
    """ Initialize the distributed environment. """
    dist.init_process_group(backend)
    fn(dist.get_rank(), dist.get_world_size())


if __name__ == "__main__":
    init_processes(run, backend='mpi')
```

Procedure

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

```
$ module load analytics
```

3. Allocate nodes in the interactive mode and execute the `start_analytics` command.

```
$ salloc -N 4
```

4. Run the Urika-XC image via the `run_training` command to execute the application using MPI:

```
$ env MPICH_MAX_THREAD_SAFETY=multiple run_training -n 32 -v --ppn 8 \
--no-node-list "python simple_mpi.py"
```

The output is similar to the following:

```
Hello from rank 28 of world size 32
Hello from rank 27 of world size 32
Hello from rank 26 of world size 32
Hello from rank 25 of world size 32
Hello from rank 30 of world size 32
Hello from rank 31 of world size 32
.….
```

The preceding examples shows a portion of the output only for brevity. The correct ranks and world size shows that native MPICH libraries are being used rather than the generic ones built in the image.

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.

   Example for allocating resources using Slurm.

```
$ salloc -N numberOfResources
```

   Example for allocating resources using PBS Pro.

```
$ qsub -I -lnodes=numberOfResources
$ module load analytics
```

3. Start an analytics cluster in development mode.

```
$ start_analytics -d
```

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 Create New Conda Environments with TensorFlow

Prerequisites

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

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_build/wheel`
- Version for CPUs and GPUs: `/opt/tensorflow_gpu_build/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 job allocation and start the analytics cluster.

   The following example is specific to Slurm

   ```
   $ salloc -N numberofNodes start_analytics
   ```

   The following example is specific to PBS Pro

   ```
   $ qsub -I -lnodes=numberofNodes
   $ module load analytics
   $ start_analytics
   ```

4. Execute the following in the analytics shell.

   ```
   $ conda create -n python2 python=2.7 pip
   $ source activate python2
   $ pip install /opt/tensorflow_cpu_build/wheel/tensorflow-1.11.0-cp27-cp27mu-linux_x86_64.whl
   ```

5. Exit the cluster.

   ```
   $ exit
   ```

6. Execute commands as needed in the new environment.

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

3.9 Train Inception-V3 Using gRPC Distributed TensorFlow

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 gRPC 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 ML PE plugin for optimal scaling of distributed TensorFlow. However, Urika-XC also provides the option to use gRPC based distributed TensorFlow instead of the Cray Programming Environment (PE) Machine Learning (ML) plugin plugin.

The `run_training` command takes one mandatory argument, namely a command 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 PE ML plugin. By default, `run_training` passes a list of comma-delimited list of nodes previously allocated by the user through their work load manager (WLM) to the command, which is responsible for using these nodes, such as, for distributed training. In case the user application does not expect these arguments, or may fail upon receiving them, the passing of node list may be suppressed by providing the command-line option `--no-node-list` to `run_training`. Refer to the `run_training` man page to learn more about using it.

3.10 Use `run_pbdR` to Execute an R MPI Application Inside an Image

About this task

About pbdR

The Urika-XC 1.2UP00 image ships with an optimized version of R, which uses OpenBLAS as the Math library. The versions of R and OpenBLAS included with the Urika-XC 1.2UP00 image are 3.4.3 and 0.2.12-1, respectively.

This optimized version of R performs better than the default one provided with the EPEL repository. R uses a set of `pbdR` packages. The `pbdR` packages included in the Urika-XC image include:

- `pbdMPI` - A high-level interface to MPI. The package handles linking issues and offers a very simple R interface for MPI programming.
- `pbdBASE` - Base utilities for distributed matrices.
- `pbdSLAP` - The Scalable Linear Algebra Package. As a distribution of `ScaLAPACK`, this package greatly simplifies package build and linking issues for distributed matrix programming.
- `pbdDMAT` - A distributed matrix of classes and methods. This package includes numerous methods for manipulating and reshaping distributed matrices, as well as linear algebra and statistics routines. Through extensive use of R's S4 methods, these functions have identical syntax to serial R.
- `pbdML` - Machine learning algorithms, using `pbdDMAT`.

- `pmclust` - Tools for parallel model-based clustering. These include k-means and Gaussian mixture modeling, and can be applied to ad-hock distributed matrices, as well as `pbdDMAT` conformable ones.
- `pbdIO` - An interface to parallel I/O packages with a focus on Single Program/Multiple Data ('SPMD') parallel programming style, which is intended for batch parallel execution.

The runtime environment of these packages can be used through the `run_pbdR` command, executes R MPI applications inside the image using the `pbdR` package. This command uses Cray MPI libraries on `pbdMPI` on Cray XC systems.

To learn more about `pbdR`, visit *https://pbdr.org/*

About the `run_pbdR` Command

The `run_pbdR` command allows the user to execute a distributed R application inside the Urika-XC image using the `pbdMPI` package. The `run_pbdR` command takes one mandatory argument, namely a command CMD, such as a run command with a R script to execute. Once provided with this mandatory argument (and possibly optional arguments), `run_pbdR` sets up the run-time environment to utilize optimized R library provided in the Urika-XC image along with the pbdR ecosystem and Cray MPI communication libraries. The user specifies the number of R 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.

This procedure can be used as a Hello World program for getting started with using the `run_pbdR` command. It uses the following sample application named `mpi_hello_world.r`:

```
# load the package
suppressMessages(library(pbdMPI, quietly = TRUE))

# initialize the MPI communicators
init()

# Hello world
message <- paste("Hello from rank", comm.rank(), "of",
comm.size())
comm.print(message, all.rank=TRUE, quiet=TRUE)

# shut down the communicators and exit
finalize()
```

Procedure

1. Log on to a login node.
2. Start up the analytics programming environment.

   ```
   $ module load analytics
   ```

3. Obtain a job allocation.

   The following example allocates 4 nodes via Slurm.

   ```
   $ salloc -N 4
   ```

4. Execute the `run_pbdR` command to run the `mpi_hello_world.r` application inside the image.

```
$ run_pbdR -n 4 "Rscript ./mpi_hello_world.r"
```

The `-n` 4 option specifies to use 4 processes, which in this case means 1 process per node. The value of the `-n` argument could be increased to run more processes per node.

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

> **NOTE:** Users should install any additional R packages needed by the application by bringing up an interactive R session using `start_analytics` and then installing new R packages to a local repository.

3.11 Run TensorFlow with the Cray Programming Environment 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-py3/1.1.4/examples/tf_mnist/mnist.py`. The script is documented with any modifications, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.1.4/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-py3/1.1.4/examples/tf_cnn_benchmarks`. Any modifications are documented inside the source files, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.1.4/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-py3/1.1.4/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-py3/1.1.4/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-9.2-v7.3/cuda/lib64$HOME/cudnn-7.1.4/lib64`

```
$ run_training -n 2 --ppn 1 --cudnn-libs /home/users/alice/CuDNN/cudnn-9.2-v7.3/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"
```

`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-9.2-v7.3/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)
```

3.12 Use PyTorch with the Programming Environment (PE) Plugin

Prerequisites

This procedure assumes that the Urika-XC software (including the Cray PE plugin) has been installed properly.

About this task

PyTorch applications can be modified to use the PE plugin for node communication. Changes that will need to be made to the application will be described in these instructions. This section is intended to provide high-level

instructions for code changes, and it will be up to the reader to adapt these instructions to their specific code base.

Procedure

1. Perform pre-training tasks.

   a. Initialize the PE plugin.

      Before any calls can be made to the PE plugin, it must be initialized. This can be achieved by calling the `mc.init_mpi()` function, and the `mc.init()` function if needed.

      ```
      mc.init_mpi()
      mc.init(num_ml_threads, num_teams, max_message_len)
      ```

   b. Broadcast the initial parameters.

      Before training begins, initial parameters should be broadcast to all ranks using the `mc.broadcast()` function from the given root rank. The initial parameters that are broadcast can be checked for accuracy using the `mc.check_buffers_match()` function.

      ```
      mc.broadcast(array_of_parameters, root_rank)
      err = mc.check_buffers_match(array_of_parameters, logging_verbosity)
      ```

   c. Initialize the thread team.

      The thread team should be initialized for gradient reduction. This can be done by calling `mc.config_team()`.

      ```
      mc.config_team(team, algorithm, ksteps, max_steps, verbosity,
      performance_frequency)
      ```

2. Update gradients during training.

   After loss has been calculated forward and backward, the gradients should be updated. One way to do this is to simply average the gradients to calculate the new gradients. In this example, we will use this method for updating gradients. After the gradients have been updated, a step should be taken for gradient descent.

   ```
   old_grads = [p.grad.data.numpy() for p in model.parameters()]
   new_grads = mc.gradients(old_grads,0)
   update_grads = [p.grad.data.numpy() for p in model.parameters()]

   # Need to assign updated averages back to p.grad.data arrays
   for update,new in zip(update_grads, new_grads):
       update[:] = new
   ```

3. Perform post-training tasks.

   After all the node communication for the application has finished, finalize the PE plugin in order to exit gracefully by calling the `mc.finalize()` function.

   ```
   mc.finalize()
   ```

3.13 Use Keras with the Cray Programming Environment (PE) Plugin

Distributed Keras applications use Horovod for node communication by default. These applications can be easily modified to use the Cray PE Plugin instead. To do this, a few changes to the code must be made. The following section describes potential modifications that need to be made to the Keras application in order to use the PE Plugin for communication. This guide is intended to provide a high-level overview of modifications to be made, and it is up to the coder to adapt these instructions to their own application.

Import Statements

`ml_comm` should be imported instead of Horovod.

```
import horovod.keras as hvd ->  import ml_comm as mc
```

Function Calls

Some changes are quite simple and only involve swapping a Horovod function call with a Plugin function call. This table shows four equivalent function calls that can be swapped.

Table 2. Function Calls

Horovod (hvd) syntax	Cray PE Plugin (mc) syntax
`hvd.init()`	`mc.init_mpi()`
`hvd.rank()`	`mc.get_rank()`
`hvd.size()`	`mc.get_nranks()`
`hvd.DistributedOptimizer`(optimizer)	`DistributedOptimizer`(optimizer)

Local Ranks

Horovod includes a function called 'local_rank' which can be called as follows:

```
hvd.local_rank().
```

This function returns a unique rank ID for the process' local position on its node. For example, assume that there are 4 nodes and 4 GPUs per node. Therefore, a total of 16 workers are spun. Each worker has a rank in the range (0-15), and a local rank in the range (0-3). There is no equivalent function available in the PE Plugin, so the ranks must be calculated in another way.

A very simple way of doing this is to only allow one process per node so that each worker's local rank on the node is 0. If this technique is used, the code modification is as follows:

```
hvd.local_rank() -> 0
```

Callbacks

Callbacks in a Keras application are used when training the model. The following example uses the `BroadcastVariablesCallback` object, but other callbacks can be used in place if they are supported.

Replace the following code:

```
callbacks = [ hvd.callbacks.BroadcastGlobalVariablesCallback(0), ]
```

with:

```
trainable_count = int(np.sum([K.count_params(p) for p in
set(model.trainable_weights)]))
ntrain_samples = x_train.shape[0]
ntest_samples = x_test.shape[0]
total_steps = int(math.ceil(epochs * (ntrain_samples + ntest_samples)/batch_size
init_plugin = InitPluginCallback(total_steps, trainable_count)
broadcast = BroadcastVariablesCallback(0)
callbacks = [init_plugin,broadcast]
```

Additionally, it may be necessary to implement the callback object. Here is an example of the implementation for the `BroadcastVariableCallback` object.

```
class BroadcastVariablesCallback(Callback):
    def __init__(self, head_rank, validate=False):
        import ml_comm as mc
        super(BroadcastVariablesCallback, self).__init__()
        self.head_rank = head_rank
        self.validate  = validate
    def on_train_begin(self, logs=None):
        sess = K.get_session()

        # Split variables based on type -> float32 vs all else
        test_v = tf.Variable([0], dtype=tf.float32)
        all_vars = tf.trainable_variables()
        float_vars = [v for v in all_vars if v.dtype == test_v.dtype]
        other_vars = [v for v in all_vars if v.dtype != test_v.dtype]

        # Initialize variables and broadcast from head node
        sess.run(tf.variables_initializer(all_vars))
        new_vars = mc.broadcast(float_vars, 0)
        bcast = tf.group(*[tf.assign(v, new_vars[k]) for k,v in
enumerate(float_vars)])
        sess.run(bcast)

        # Validate Broadcast
        if self.validate:
            py_all_vars = [sess.run(v) for v in float_vars]
            var_types = [np.array([v]) if type(v) == np.float32 else v for v in
py_all_vars]
            if mc.get_rank() is 0:
                if (mc.check_buffers_match(var_types, 1) != 0):
                    tf.logging.error("Not all processes have the same initial
model!")
                else:
                    tf.logging.info("Initial model is consistent on all ranks")
```

Command Line

To run the modified script, a few changes must be made to the command line arguments passed to the `run_training` script.

1. Using the `--craype-plugin-libs` flag, pass in the directory where the Cray PE plugin is located

2. Using the `--craype-plugin-version` flag, pass in the version of the Cray PE plugin being used
3. (OPTIONAL): If using the technique described in the "Local Ranks" section to assign local ranks, ensure that the `--ppn` flag is set to 1

This should now run the application using the Cray PE Plugin for communication, instead of using Horovod.

3.14 Run Keras Using the Tensorflow Backend

Prerequisites

This procedure assumes that Slurm is used as the system's workload manager.

About this task

Keras is a high-level neural networks API, written in Python and designed to simplify the building neural networks. It can be run on top of TensorFlow, CNTK, and Theano. Low-level operations such as tensor products, convolutions, etc., use the backend engine or framework. The default backend used is Tensorflow.

The Keras GitHub repository includes a set of examples that can be used for getting started with Keras. These examples can be found at *https://github.com/keras-team/keras/tree/master/examples*. For more information, refer to *https://keras.io/*.

This procedure provides instructions for setting up Keras on an XC system and using it from Tensorflow CPU and GPU Conda environments.

Procedure

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

   ```
   $ module load analytics
   ```

3. Obtain an allocation on Slurm and execute the `start_analytics` command.

   ```
   $ salloc -N 4 start_analytics
   ```

4. Export the path to Python by selecting one of the following options:
   - To use Keras with a CPU, execute:

     ```
     $ export PYTHONPATH=/opt/tensorflow_cpu:$PYTHONPATH
     ```

   - To use Keras with a GPU, execute:

     ```
     $ export PYTHONPATH=/opt/tensorflow_gpu:$PYTHONPATH
     ```

5. Verify that Keras has been set up correctly by running some Keras tests.

   ```
   $ python -c "import tensorflow"
   ```

If the command above fails, check if `pythonpath` is set correctly and if Tensorflow has been installed properly. If Tensorflow is working correctly, re-check the `pythonpath` and check if there are any issues related to installation of Keras.

6. Verify that Keras is working.

```
$ python -c "import keras"
Using TensorFlow backend
```

The text, 'Using TensorFlow backed' indicates that Keras is functioning properly.

3.15 Run Horovod with Tensorflow, Keras, or PyTorch

Prerequisites

This procedure assumes that Slurm is being used as the system's workload manager.

About this task

Horovod is an distributed training framework, through which deep learning models can be trained across Cray XC systems. Horovod is built to use Cray MPI on XC systems. A version of Horovod that is optimized for CPU and GPU has been installed at `/opt/horovod_cpu` and `/opt/horovod_gpu`, respectively. The GPU version is built with NCCL2 support. Given that XC system GPU nodes have only one GPU, Cray MPI is used for cross node communication. When Urika-XC is launched, Cray MPI libraries are mounted at the `/opt/cray_mpi` directory inside the image. Hence these libraries are assumed to be available on the system.

This procedure provides instructions for running Horovod via the `run_training` command. The user application is assumed to use Horovod internally.

Procedure

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

```
$ module load analytics
```

3. Run Horovod using the `run_training` command.
   - To run a Tensorflow model using Horovod on a CPU system, execute:

```
$ run_training --no-node-list "python \
./benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
--model=resnet50 --num_intra_threads=32 --num_inter_threads=2 \
--num_batches=20 --batch_size=32 --variable_update=horovod \
--train_dir=$HOME/tf_cnn_train/"
```

   - To run a Tensorflow model using Horovod on a GPU system, execute:

```
$ run_training --no-node-list "python \
./benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
--model=resnet50 --num_intra_threads=32 --num_inter_threads=2 \
```

```
--num_batches=20 --batch_size=32 --variable_update=horovod \
--train_dir=$HOME/tf_cnn_train/" --cudnn-libs CuDNN/cuda/lib64
```

On a GPU system, the path location of CuDNN libraries to the GPU optimized Horovod is used.

When using `start_analytics` to interactively run an application, launch the image, and then set `PYTHONPATH` to point to the Horovod installation as shown below:

1. For a CPU based system, execute:

```
$ export PYTHONPATH=/opt/tensorflow_cpu:/opt/horovod_cpu/${URIKA_GNU_VER}:
$PYTHONPATH
```

2. For a GPU based system, execute:

```
$ export PYTHONPATH=/opt/tensorflow_gpu:/opt/horovod_gpu/${URIKA_GNU_VER}:
$PYTHONPATH
```

3.16 Select GNU Version

The Cray Machine Learning (ML) Programming Environment (PE) Plugin has two variants that are compatible with Cray MPI GNU 5 and GNU 7. These variants include:

- `gnu53`
- `gnu71`

Similarly, Horovod also has two variants that support Cray MPI GNU 5 and Cray MPI GNU 7.

For Cray PE 2.5.16.31, (which includes MPT 7.7.3), Urika XC 1.2UP00 uses GNU 5 of the Cray MPI libraries, in order to run the Cray ML PE Plugin and Horovod.

MPT versions older than 7.7.3 do not support GNU 5. If it is required to run these features using an upgraded Cray PE environment that has a more recent version of MPT than 7.7.3, set the value of `URIKA_GNU_VER` to 7.1 before launching `run_training` or `start_analytics` commands, as shown below:

```
URIKA_GNU_VER=7.1
```

3.17 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.

   Example for Slurm:

```
$ salloc -N numberofNodes
```

Example for PBS Pro:

```
$ qsub -I -lnodes=numberofNodes
$ module load analytics
```

3. Load the `analytics` module

```
$ module load analytics
```

4. Create an SSH tunnel from the localhost to the 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.

5. 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.

```
$ 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.

For Slurm, allocating resources and starting the analytics cluster can be performed in one go, as shown in the following example:

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

6. 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
```

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.

   Alternatively, set the password in the Jupyter Notebook server.

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

3.17.1 Run R in a Jupyter Notebook with the IRKernel Package

About this task

The IRKernel package enables running R in Jupyter Notebook. IRkernel is not pre-installed on the Urika-XC image by default. This procedure can be used to install it in the image in order to use R in Jupyter Notebook.

Procedure

1. Bring up an interactive R session using `start_analytics`

2. Install the following R packages to a local repository
   by executing the following from the R console:

```
>install.packages(c('repr', 'IRdisplay', 'evaluate', 'crayon', 'pbdZMQ',
'devtools', 'uuid', 'digest'))
>devtools::install_github('IRkernel/IRkernel')
```

3. Execute the following command from the R console to enable Jupyter to access the installed R kernel by installing a kernel specification.

```
> IRkernel::installspec()
```

4. Start up a Jupyter Notebook by bringing up an interactive session.

5. Ensure that the packages were installed correctly by verifying that the Jupyter Notebook's drop down listing available kernels contains an option for starting up a new notebook using R.

   For more information, refer to *https://irkernel.github.io/running/* and *https://irkernel.github.io*.

3.18 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. Load the analytics module.

```
$ module load analytics
```

2. Allocate resources.

   Example for Slurm:

```
$ salloc -N numberOfNodes
```

   Example for PBS Pro:

```
$ qsub -I -lnodes=numberOfNodes
$ module load analytics
```

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

   - Slurm:

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

     PBS Pro:

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

     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 from the laptop being used to the login node port on the login node.

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

   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 CS_HOST_NAME
```

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. Similarly, conflicts with users running Jupyter Notebook or other web-based UIs need to be resolved as well. 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.

4 Set up Connectivity

4.1 Set up Connectivity to User Interfaces

About this task

An SSH tunnel can be useful for connecting to a UI running on the interactive node from a different machine. 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 a port on the user's machine, such as a laptop, that will be used to view the UI locally.
- *`loginPort`* is the login node of the system.
- *`UIport`* is the port on the interactive node that the web UI runs on.

Procedure

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

   ```
   $ module load analytics
   ```

3. Allocate resources.

   Example for Slurm:

   ```
   $ salloc -N numberofNodes
   ```

   Example for PBS Pro:

   ```
   $ qsub -I -l nodes=numberofNodes
   $ module load analytics
   ```

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

   Example for Slurm:

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

   Example for PBS Pro:

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

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.

5. Create an SSH tunnel from the localhost to the 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.

4.2 Set up Connectivity 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.

5 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.

5.1 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.

5.2 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. Load the CGE module.

```
$ module load cge
```

4. 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

5. 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

6. 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

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

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

The results produced by the browser may not appear the same as the results seen previously in step 6, depending on the output format chosen (or automatically selected). However, the same results will be encoded.

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 &
```

8. 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.

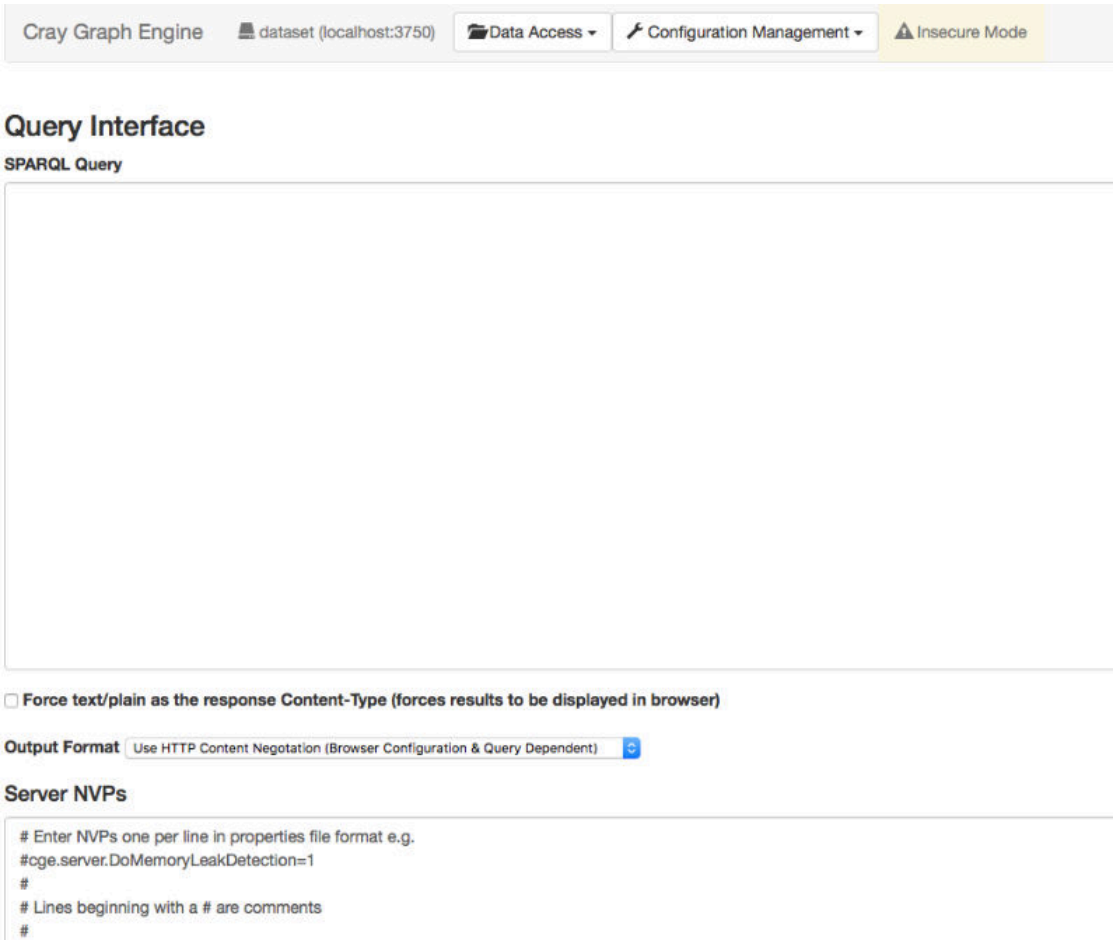
When the CGE front end has been launched, a message similar to the following will be returned on the command-line:

```
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

9. 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 .
}
```

When the query finishes executing the results will either displayed in your browser or downloaded by your browser to your default download directory. This will depend on both the browser used and the output format selected, which may be automatic.

CGE Front End Termination

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

There is no output returned by the system when the CGE front end is terminated.

CGE Server Shutdown

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

```
$ cge-cli shutdown
```

6 Hyperparameter Optimization (HPO) Support

Machine and deep learning algorithms require a significant amount of intuition and guesswork to design solutions to new problems. Hyperparameter optimization is a way to remove this guesswork from many of the design decisions that go into a model.

Traditional HPO techniques can sweep or optimize over any model structure parameter, such as the number of neural network layers, the size of those layers, activation functions, and many others. Additionally, training parameters such as learning rate, weight decay, and dropout can be searched and optimized by both traditional HPO and population based training.

HPO typically works in generations. These generations are populated by unique sets of hyperparameters to be evaluated. The evaluation is generally the final loss value for the trained network, accuracy, or some other metric to be minimized. When each member of the generation has been evaluated, a new generation is populated by the underlying algorithm. This can happen in either a brute force way, which would likely target smaller models, or an optimized way that would target a larger model, where a complete search of possible hyperparameters would be simply too resource intensive.

About the `crayai` Module

As part of this release of Urika-XC, Cray is providing early access to our HPO package accessed through the `crayai` Python module. This module is activated as part of the typical analytics packages, and can be run independently of the Urika-XC `run_training` and `start_analytics` commands.

`crayai` exposes two potential options for distribution of hyperparameter evaluation. For those running with Slurm as a workload manager, `crayai` can interface directly with Slurm running natively on the login nodes. If the training process depends on the packages supported by the analytics image, such as Tensorflow or PyTorch, Urika-XC can also be used as the launcher. In this case, all three supported workload mangers supported by Urika-XC are supported, Additionally, all evaluations will run within Urika-XC Shifter containers, which will retain a consistent development environment, while allowing customizations through Anaconda Python.

Supported HPO Techniques

The following HPO techniques are supported on Urika-XC:

- **Genetic** - The Cray genetic HPO algorithm is an optimization technique. It relies on a genetic machine learning algorithm to learn ideal sets of hyperparameters, based on prior evaluations. This happens by treating the final loss or accuracy value returned to the `crayai` HPO submodule as a "Feature of Merit", which is then used to judge to quality of those hyperparameters. Based on this judgement, poor performing hyperparameter are pruned and successful hyperparmaters are "mutated" or augmented by a small factor and "crossover" is applied where the hyperparameters from two individuals are combined to create a new individual.
- **Random** - The random HPO algorithm is a simple hyperparameter search technique that relies on brute force and random chance to find better combinations of hyperparamters.

- **Grid** - The grid HPO algorithm is another simple hyperparameter search technique that relies on a more methodical sweep of a defined search space. An N-dimensional matrix is defined based on the hyperparameters provided and each element is evaluated.
- **Population Based Training (PBT)** - Population based training is a specific application of an HPO algorithm with the intent of intelligently learning a schedule for training parameters, such as learning rate and weight decay. Typically, these schedules are set similar to other hyperparameters, using intuition, trial, and error.

  Population based trainings allow these values to vary from update to update among some number of candidates. At the end of a training window, these candidates are evaluated, and by using the genetic approach above, the lowest performing candidates are dropped and the best performing ones are mutated and augmented. Unlike traditional HPO, after evaluation, the training continues through a checkpoint model with a fresh batch of candidate parameters. By setting this window properly, parameters, such as learning rate, can vary as necessary based on the current training environment. Due to the dependence on checkpointing and restoring models, the model structure must remain static during a training process.

  With Cray's distributed HPO framework and sufficient hardware resources, population based trainings can distribute the evaluation of candidate training parameters, leading to a total PBT training time that is similar to what it would take to run a single training process, but with significantly better results in most cases than would be achieved by setting heuristic training parameter schedules.

6.1 Get Started with Using Hyperparameter Optimization (HPO)

Prerequisites

This procedure requires the following software to be installed on the system:

- Python 3.3 or greater
- numpy

About this task

To use the HPO framework, a user must perform the following steps:

1. Import the required module.
2. Define parameters to be optimized.
3. Create an Evaluator.
4. Create an Optimizer.
5. Optimize over the parameter.

Descriptions of each of these steps is provided in this procedure.

Procedure

1. Load the desired Python 3 environment.

   The site default `python3` or a local `python3` environment can be used as long as `numpy` is installed.

   ```
   $ module load cray-python
   ```

2. Import the `hpo` submodule of the `crayai` module in a Python script.

```
from crayai import hpo
```

3. Define parameters to be optimized.

   These are the parameter to optimize over. They are exposed to the training program through command-line flags. The `crayai hpo` tool searches within a specified range, starting at a specified default value.

   Hyperparameter Definition Format

```
params = hpo.Params([[command_line_flag_1, default_val_1, (min_val, max_val)],
                     [command_line_flag_2, default_val_2, (min_val, max_val)],
                     ...
                    ])
```

   Hyperparameter Definition Example

```
params = hpo.Params([["--learningRate", 0.01, (1e-6, 1.0)],
                     ["--neuronsPerLayer", 100, (50, 500)],
                     ["--dropoutRate", 0.5, (0.0, 0.7)]])
```

4. Create an Evaluator.

   The Evaluator class defines how to evaluate a set of hyperparameters by running the kernel program (model training script) with command-line arguments. This includes distribution of individual evaluations via a workload manager (specified as `wlm`), the Urika-XC launcher (specified as `urika`), or local mode (specified as `none`). The Evaluator can handle distributed training processes via the `nodes_per_eval` parameter, and can calculate the number of parallel evaluations that can be executed simultaneously within the given allocation.

   Evaluator Definition Format

```
evaluator = hpo.Evaluator(command,        # Command to run to evaluate the hyperparameters
                          run_path,       # Opt: Workspace directory for log files.
                          fom,            # Opt: Unique string identifying where field of
                                          #      merit value will be in evaluation output.
                          checkpoint,     # Opt: Path to checkpoint directory per workspace.
                          alloc_job_ID,   # Opt: Allocation id for existing allocation
                                          #      (wlm launcher only)
                          nodes,          # Opt: Total node count in the allocation
                          nodes_per_eval, # Opt: Nodes needed for each evaluation
                          launcher,       # Opt: How to distribute the evaluation. Choose
                                          #      from "urika", "wlm", or "none"
                          urika_args,     # Opt: Argument to pass on to run_training for
                                          #      the urika launcher
                          verbose)        # Opt: Verbose print message.
```

   Evaluator Example

```
cmd = "python source/train.py --epochs 5"
evaluator = hpo.Evaluator(cmd,
                          nodes=8,
                          nodes_per_eval=2
                          launcher='urika',
                          urikaArgs="--no-node-list",
                          verbose=False)
```

   In the preceding example:

   - The training process defined in `source/train.py` will run with 5 full epochs every time it is executed.
   - The Evaluator will have access to 8 nodes in an allocation.
   - Each evaluation will run on 2 nodes, allowing 4 evaluations to occur in parallel.

- The `urika` launcher will be used to run the command with `run_training` from the Urika-XC package.
- `--no-node-list` will be passed as an additional argument to `run_training` for each evaluation.
- Verbose logging information will not be printed.

5. Create an Optimizer.

The Optimizer contains the core algorithms behind HPO, specifically genetic, random and grid searches. The Optimizer works in tandem with the Evaluator by ingesting the results from the Evaluator and returning a new set of hyperparameters to be evaluated.

Optimizer Definition Format

```
optimizer = hpo.genetic.Optimizer(evaluator, # Evaluator instance
                                  generations,        # Opt: Number of generations.
                                  num_demes,          # Opt: Number of distinct demes (populations)
                                  pop_size,           # Opt: Number of individuals per deme
                                  mutation_rate,      # Opt: Probability of mutation per
                                                      #      hyperparameter during creation of next
                                                      #      generation
                                  crossover_rate,     # Opt: Probability of crossover per
                                                      #      hyperparameter during creation of next
                                                      #      generation
                                  migration_interval, # Opt: Interval of migration between demes
                                  log_fn,             # Opt: Filename to record results of
optimization
                                  verbose)            # Opt: Enable verbose output
optimizer = hpo.random.Optimizer(evaluator, # Evaluator instance
                                 numIters,  # Opt: Number of iterations to run
                                 seed,      # Opt: Seed for random number generator. Defaults to 0,
                                            #      i.e. random seed used.
                                 verbose)   # Opt: Enable verbose output
optimizer = hpo.grid.Optimizer(evaluator,  # Evaluator instance
                                 grid_size,  # Opt: Number of grid points to discretize for each
                                             #      hyperparameter
                                 chunk_size, # Opt: Number of grid points to evaluate per batch
(chunk)
                                 verbose)    # Opt: Enable verbose output
```

Optimizer Example

```
optimizer = hpo.genetic.Optimizer(evaluator,
                                   pop_size= 4,
                                   num_demes=2,
                                   generations=5,
                                   mutation_rate=0.10,
                                   crossover_rate=0.4,
                                   verbose=True )

optimizer.optimize(params)
```

6.2 Hyperparameter Optimization (HPO) Examples

This section provides examples of using HPO with MNIST, PBT, and Distributed trainings.

MNIST Training

Training Pseudocode - Contents of a sample script named `train_mnist.py` are shown below:

```
import tensorflow as tf
import argparse

# Expose hyperparamters through commandline arguments
```

```
argparser = argparse.ArgumentParser()
argparser.add_argument("--learning_rate", type=float, default=0.001)
argparser.add_argument("--dropout_rate", type=float, default=0.5)
argparser.add_argument("--num_layers", type=int, default=2)
argparser.add_argument("--num_epochs", type=int, default=5)
args = argparser.parse_args()

# Get dataset...
data = MNIST_Data(...)

# Define model with number of layers provided at the command line
model = MNIST_Model(nlayers = args.num_layers)
# Add dropout at the rate provided at the command line
model.add_dropout(rate=args.dropout_rate)

# Define a stochastic gradient decent optimizer in tensorflow
opt = tf.train.GradientDescentOptimizer(learning_rate=args.learning_rate)

# Train...
total_loss, accuracy = model.train(data, optimizer=opt, epochs=args.num_epochs)

# Print the feature of merit in a form that will be recognized by the hpo tool
print("FoM: %e" % total_loss)
```

Sample `mnistHPO.py` Wrapper Script

```
# Import the hpo submodule
from crayai import hpo


# Define hyperparameter and ranges
params = hpo.Params([["--learningRate", 0.01, (1e-6, 1.0)],
                     ["--num_layers", 5, (1, 12)],
                     ["--dropoutRate", 0.5, (0.0, 0.7)]])

# Define the evaluator
cmd = "python source/train_mnist.py --num_epochs 5"
evaluator = hpo.Evaluator(cmd,
                          nodes=8,
                          launcher='urika',
                          urika_args="--no-node-list",
                          verbose=False)

# Define the optimizer
optimizer = hpo.genetic.Optimizer(evaluator,
                                  pop_size= 16,
                                  num_demes=2,
                                  generations=25,
                                  mutation_rate=0.10,
                                  crossover_rate=0.3,
                                  verbose=True)

# Run the optimizer over the provided hyperparameters
optimizer.optimize(params)
```

PBT Training

Training Pseudocode - Contents of a sample script named `train_mnist.py` are shown below. Users can add options for saving the model after training and loading a model from a prior checkpoint. In addition, it is also useful to have a command line argument for total number of epochs to compute.

```
import tensorflow as tf
import argparse

# Expose hyperparamters through commandline arguments
# PBT: Add command line arguments for saving and loading checkpoints.
argparser = argparse.ArgumentParser()
argparser.add_argument("--learning_rate", type=float, default=0.001)
argparser.add_argument("--dropout_rate", type=float, default=0.5)
argparser.add_argument("--num_layers", type=int, default=2)
argparser.add_argument("--num_epochs", type=int, default=5)
argparser.add_argument('--load_checkpoint', type=str)
argparser.add_argument('--save_checkpoint', type=str, default='')
args = argparser.parse_args()

# Get dataset...
data = MNIST_Data(...)

# Define model with number of layers provided at the command line
model = MNIST_Model(nlayers = args.num_layers)
# Add dropout at the rate provided at the command line
model.add_dropout(rate=args.dropout_rate)

# PBT: Load weights from checkpoint file
if args.load_checkpoint != None:
    model.load_weights(args.load_checkpoint)

# Define a stochastic gradient decent optimizer in tensorflow
opt = tf.train.GradientDescentOptimizer(learning_rate=args.learning_rate)

# Train...
total_loss, accuracy = model.train(data, optimizer=opt, epochs=args.num_epochs)

# PBT: Save a checkpoint
if args.save_checkpoint != None:
    model.save_weights(args.save_checkpoint)

# Print the feature of merit in a form that will be recognized by the hpo tool
print("FoM: %e" % total_loss)
```

Sample `mnistPTB.py` Wrapper Script - In the following sample script:

- Remove `num_layers` as a hyperparameter, since PBT requires checkpointing and a consistent model architecture.
- Set number of `epochs` to 1. Alternatively, set the number of steps to an appropriate number for a more fine-grained training schedule

```
# Import the hpo submodule
from crayai import hpo

# Define hyperparameter and ranges
# PBT: Remove model architecture hyperparameters such as num_layers
params = hpo.Params([["--learningRate", 0.01, (1e-6, 1.0)],
                     ["--dropoutRate", 0.5, (0.0, 0.7)]])

# Define the evaluator
# PBT: Add to the command the arguments for checkpointing the model.
#      Note the @checkpoint which will be used by the evaluator to
#      set the proper running/checkpoint directory given by the
#      checkpoint option when defining the Evaluator object.
```

```
cmd = "python source/train_mnist.py --num_epochs 1 " + \
      "--load_checkpoint=@checkpoint/model.h5 "      + \
      "--save_checkpoint=@checkpoint/model.h5 "
checkpoint_dir = "./checkpoints"
evaluator = hpo.Evaluator(cmd,
                          checkpoint=checkpoint_dir,
                          nodes=8,
                          launcher='urika',
                          urika_args="--no-node-list",
                          verbose=False)

# Define the optimizer
# PBT: Add optional parameter gen_per_epoch, allowing more
#      optimization for each training parameter per epoch
optimizer = hpo.genetic.Optimizer(evaluator,
                                  gens_per_epoch=5,
                                  pop_size= 16,
                                  num_demes=2,
                                  generations=25,
                                  mutation_rate=0.10,
                                  crossover_rate=0.3,
                                  verbose=True)

# Run the optimizer over the provided hyperparameters
optimizer.optimize(params)
```

Distributed Training

Training Pseudocode - Contents of a sample script named `train_mnist_dist.py` are shown below. Note that this is a simplified training script for illustration purposes only. It cannot be executed as-is.

```
# Dist: Import the Cray-PE Machine Learning Plugin
import ml_comm as mc
import tensorflow as tf
import argparse

# Expose hyperparamters through commandline arguments
# PBT: Add command line arguments for saving and loading checkpoints.
argparser = argparse.ArgumentParser()
argparser.add_argument("--learning_rate", type=float, default=0.001)
argparser.add_argument("--dropout_rate", type=float, default=0.5)
argparser.add_argument("--num_layers", type=int, default=2)
argparser.add_argument("--num_epochs", type=int, default=5)
argparser.add_argument('--load_checkpoint', type=str)
argparser.add_argument('--save_checkpoint', type=str, default='')
args = argparser.parse_args()

# Get dataset...
data = MNIST_Data(...)

# Define model with number of layers provided at the command line
model = MNIST_Model(nlayers = args.num_layers)
# Add dropout at the rate provided at the command line
model.add_dropout(rate=args.dropout_rate)

# Dist: Initialize the ML plugin
mc.init(1, 1, 20*1024*1024, "tensorflow")

# Dist: Broadcast weight initialization to sync random weight generation between
nodes
```

```
session = tf.get_session()
new_vars   = mc.broadcast(tf.trainable_variables(),0)
bcast = tf.group(*[tf.assign(v,new_vars[k]) for k,v in
enumerate(tf.trainable_variables())])
session.run(bcast)

# PBT: Load weights from checkpoint file
if args.load_checkpoint != None:
    model.load_weights(args.load_checkpoint)

# Define a stochastic gradient decent optimizer in tensorflow
# Dist: Break out optimization step to aggregate gradients between ranks
opt = tf.train.GradientDescentOptimizer(learning_rate=args.learning_rate)
grads_and_vars = optimizer.compute_gradients(model.loss)
grads = mc.gradients([gv[0] for gv in grads_and_vars], 0)
agg_grads_and_vars = [(g,v) for (_,v), g in zip(grads_and_vars, grads)]
train_op = optimizer.apply_gradients(gs_and_vs)

# Train...
total_loss, accuracy = model.train(data, optimizer=train_op,
epochs=args.num_epochs)

# PBT: Save a checkpoint
# Dist: Only save weight from rank 0
if args.save_checkpoint != None and mc.get_rank() == 0:
    model.save_weights(args.save_checkpoint)

# Print the feature of merit in a form that will be recognized by the hpo tool
# Dist: Only print from rank 0 to clear up output
if mc.get_rank() == 0:
    print("FoM: %e" % total_loss)
```

Sample `mnistDistPTB.py` Wrapper Script

```
# Import the hpo submodule
from crayai import hpo

# Define hyperparameter and ranges
# PBT: Remove model architecture hyperparameters such as num_layers
params = hpo.Params([["--learningRate", 0.01, (1e-6, 1.0)],
                     ["--dropoutRate", 0.5, (0.0, 0.7)]])

# Define the evaluator
# PBT: Add to the command the arguments for checkpointing the model.
#      Note the @checkpoint which will be used by the evaluator to
#      set the proper running/checkpoint directory given by the
#      checkpoint option when defining the Evaluator object.
# Dist: Add the option 'nodes_per_eval' to indicate that the evaluator
#      should allocate 2 nodes for each individual evaluation.
#      Note that this will decrease the number of parallel evaluations,
#      but also will decrease training time per evaluation.
cmd = "python source/train_mnist.py --num_epochs 1 " + \
      "--load_checkpoint=@checkpoint/model.h5 "      + \
      "--save_checkpoint=@checkpoint/model.h5 "
checkpoint_dir = "./checkpoints"
evaluator = hpo.Evaluator(cmd,
                          checkpoint=checkpoint_dir,
                          nodes=8,
                          nodes_per_eval=2,
                          launcher='urika',
```

```
                          urika_args="--no-node-list",
                          verbose=False)

# Define the optimizer
# PBT: Add optional parameter gen_per_epoch, allowing more
#      optimization for each training parameter per epoch
optimizer = hpo.genetic.Optimizer(evaluator,
                                  pop_size= 16,
                                  num_demes=2,
                                  generations=25,
                                  mutation_rate=0.10,
                                  crossover_rate=0.3,
                                  verbose=True)

# Run the optimizer over the provided hyperparameters
optimizer.optimize(params)
```

Expected Output

Sample output of running the preceding wrapper scripts is shown below:

```
$ salloc -N 8 --exclusive
salloc: Granted job allocation 60587
salloc: Waiting for resource configuration
salloc: Nodes nid000[1-8] are ready for job

$ module load analytics
$ module load cray-python
$ python mnistHPO.py
------------------------------------------------------------
Settings:
------------------------------------------------------------
generations:       25
numDemes:          2
popSize:           16
verbose:           true
mutationRate:      0.1
crossoverRate:     0.3
migrationInterval: 5
------------------------------------------------------------
Locales: 1
------------------------------------------------------------
------------------------------------------------------------
Adding 32 individuals to each deme with genotype:
--dropout_rate:  5.000000e-01,
--learning_rate:  1.000000e-01,
--num_layers:  5.000000e+00,
Adding mutants to first generation.
------------------------------------------------------------
------------------------------------------------------------
Generation: 0
...
```

6.3 Hyperparameter Optimization (HPO) Troubleshooting Information

Verbose Mode

Verbose options of the evaluator and optimizer provide insight into different portions of the HPO process. By setting the evaluator to verbose mode, information regarding the distribution of evaluations and any `stderr` coming from those evaluations is printed to the console. By setting the optimizer to verbose mode, details from the optimization process are printed between generations, providing insight into how the algorithm is adjusting the hyperparameters. Details from the optimization process also indicate progress towards minimizing the feature of merit.

Suggested Debugging Steps

- Ensure the produced evaluator commands run properly outside of the HPO tool.
- Set the evaluator to verbose mode.
  - Run the command with a separate `run_training` or `srun` command, starting with a single node and working up for distributed trainings.
  - Ensure that the Urika-XC commands are working as expected when `urika_args` is provided to the Evaluator
- Try setting `-v` in the `urika_args` parameter, or when running outside of the HPO tool, as illustrated in the following example:

```
evaluator = hpo.Evaluator(cmd, nodes=8, launcher='urika', urika_args="-v --no-node-list", verbose=False)
```

7 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 restarted 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 3. 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 4. Urika-XC Software Component Versions

Software Component	Version
CGE	3.2UP04
Apache Spark	2.3.2
Anaconda Distribution of Python	5.2.0
Dask	0.14.3 and later
Dask distributed	1.21 and later
Intel BigDL	0.7.0
Keras	2.2.4
PyTorch	1.0.0
Horovod	0.15.2
HPO	0.1
Cray Machine Learning (ML) Programming Environment (PE) plugin	1.1.4
pbdR	pbdR contains a number of packages, including: • `pbdIO_0.1-0` • `pbdML_0.1-2` • `pbdSLAP_0.2-4` • `pmclust_0.2-0` • `pbdDMAT_0.5-0` • `pbdBASE_0.5-0` • `pbdMPI_0.3-8`
Analytics Programming Environment	
Python	3.6 as part of Anaconda 5.2.0. Anaconda also supports creating python environments with 2.7, 3.4, and 3.5
Java	1.8
Scala	2.11.8
R	3.5.1
Maven	3.3.9
SBT	0.13.9
ANT	1.9.2
TensorFlow	1.11

Software Component	Version
TensorBoard	1.11
Jupyter NoteBook	4.3.0
CuDNN	7.3.1
CUDA	9.2
MKL	0.16

Environment Variables

Table 5. Environment Variables and Mappings

Environment Variable	Mapping
ANACONDA_DIR	/opt/anaconda
JAVA_HOME	/usr/lib/jvm/jre-1.8.0
MAVEN_HOME	/usr/share/apache-maven
SPARK_VERSION	2.2.0
SPARK_HADOOP_VERSION	2.7
SPARK_DIR	/usr/spark
SCALA_VERSION	2.11.8
BIGDL_VERSION	0.7.0
BIGDL_DIR	/opt/bigdl-0.7.0/dist
BIGDL_JAR	/opt/bigdl-0.7.0/dist/lib/bigdl-0.7.0-jar-with-dependencies.jar
SPARK_WORKER_PORT	8888
DASK_WORKER_PORT	19866
DASK_NANNY_PORT	19868
DASK_BOKEH_PORT	19870
BAZEL_PATH	/opt/bazel-0.5.4
MKL_DNN_PATH	/opt/mkl-dnn-v0.10

LD_LIBRARY_PATH	The following path has been split into two lines because of lack of space.
	`/opt/mkl-dnn-v0.10/lib:/opt/cudnn:/usr/local/lib:/usr/lib/ \`
	`x86_64-linux-gnu:/usr/local/lib:/usr/spark/mathlibs`

PATH	The following path has been split into two lines because of lack of space.
	`/opt/rt_scripts/bin:/opt/anaconda/bin:/usr/local/sbin:\`
	`/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`

7.1 BigDL Logging

BigDL implements a method named `redirectSparkInfoLogs`, which 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.