Urika®-CS Analytic Applications Guide

(1.0.UP00)

S-3024

Contents

1 About the CS™ Series Urika®-CS AI and Analytics Applications Guide

The *CS™ Series Urika®-CS AI and Analytics Applications Guide* includes procedures for using the Urika®-CS software stack on Cray CS systems. It provides an overview of the Urika-CS software stack, information about using the various OSA components and the Cray PE ML plugin, as well as troubleshooting and quick reference information.

Table 1. Record of Revision

Publication Title	Date	Release
CS™ Series Urika®-CS AI and Analytics Applications Guide	July 2018	Urika-CS 1.0UP00

Scope and Audience

This publication is written for administrators and end users of the Urika®-CS software.

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

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, Urika-GX, and YARCDATA. The following are trademarks of Cray Inc.: APPRENTICE2, CHAPEL, CLUSTER CONNECT, ClusterStor, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

2 About Urika®-CS

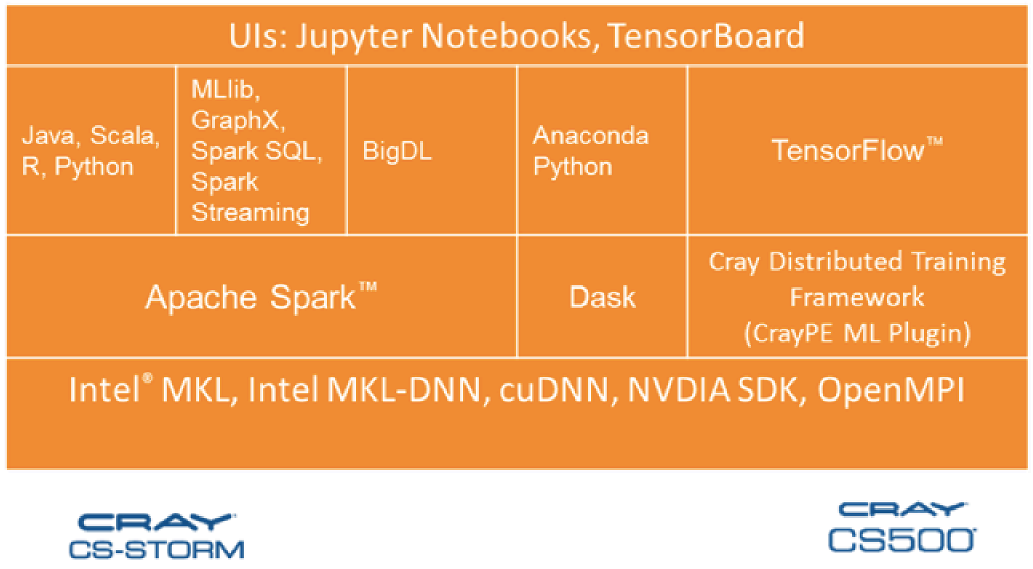
Cray Urika®-CS is an optimized big data software stack, which is tuned for multiple work-flows and runs on Cray CS systems. It features a comprehensive analytics software stack for performing machine and deep learning tasks.

Features and Analytic Components

- **Cray PE ML Plugin** - This portable plugin leverages features of MPI as well as a novel "delayed synchronization" variant of the Stochastic Gradient Descent (SGD) algorithm to allow scaling of deep learning training without the normal convergence penalties. These capabilities improve scale, performance, and ease of use of work loads.
- **Support for Jupyter Notebook** - Jupyter Notebook is a web application that enables creating and sharing documents that contain live code, equations, visualizations, and explanatory text. For more information, visit *http://jupyter.org*
- **Support for GPUs** - Urika®-CS enables running TensorFlow on Nvidia GPU nodes.
- **Open Source Analytics (OSA) Images -** Urika®-CS provides Open Source Analytics (OSA) OSA images that run inside Singularity 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®-CS software ships with a number of Spark ecosystem components. For more information, visit *https://spark.apache.org*
  - **Anaconda® Python and R -** Anaconda is a distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing. It aims at simplifying package management and deployment. For more information, visit *https://anaconda.org*
  - **Dask and Dask Distributed** - Dask is a parallel programming library that combines with the Numeric Python ecosystem to provide parallel arrays, data-frames, machine learning, and custom algorithms. For more information, visit *http://dask.pydata.org*
  - **Intel® BigDL** - BigDL is a distributed deep learning library for Spark that can run directly on top of existing Spark or Apache Hadoop clusters. Deep learning applications can be written as Scala or Python programs. For more information, visit *https://www.intel.com*
  - **TensorFlow™ and TensorBoard** - TensorFlow is a software library for dataflow programming across a range of tasks. It is a Math library, which is also used for machine learning applications, such as neural networks. TensorFlow provides a utility called TensorBoard that can display an interactive graphic of the computational graph. For more information, visit *https://www.tensorflow.org*

  Urika®-CS's software stack is depicted in the following figure:

Figure 1. Urika®-CS Analytic Software Stack


UIs: Jupyter Notebooks, TensorBoard
Java, Scala, R, Python
MLlib, GraphX, Spark SQL, Spark Streaming
BigDL
Anaconda Python
TensorFlow™
Apache Spark™
Dask
Cray Distributed Training Framework (CrayPE ML Plugin)
Intel® MKL, Intel MKL-DNN, cuDNN, NVDIA SDK, OpenMPI
CRAY CS-STORM
CRAY CS500

2.1 About Open Source Analytics (OSA) Images

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

Only OSA images can be used as part of Urika-CS software. Software packages, such as Spark and Dask, etc. are available inside the OSA image to enable building distributed AI pipelines.

CAUTION: Adding new containers or modifying the containers shipped with the Urika-CS software is currently not supported. Similarly, Cray does not currently support customer modification to the Urika-CS release. For more information, please contact a Cray service representative.

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

2.2 Resource Allocation

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:

- Slurm, *N*-2 worker containers will be launched.
- PBS Pro, *N*-2 worker containers will be launched.

2.3 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-CS ships with Spark 2.2.0.

Run Spark Applications

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

- `spark-shell`
- `spark-submit`
- `spark-sql`
- `pyspark`
- `sparkR`
- `run-example`

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

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

Build Spark Applications

Spark 2.2.0 builds with Scala 2.11.8.

Urika-CS 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" % "2.2.0"
```

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

```
<dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.11</artifactId>
      <version>2.2.0</version>
    </dependency>
</dependencies>
```

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

Conda Environments

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

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-CS 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.

2.3.1 Local Storage Options for Apache Spark

On Urika-CS, Spark in Singularity containers will default to looking for the `/tmp` directory on the local disk for scratch space/local storage data.

It is important to note the following items when using Spark in Singularity containers:

- If local storage exists and is mapped to `/tmp`, then no change required to the Singularity configuration.
- If local storage exists but `/tmp` is not mapped to the local storage, administrators can create a local directory and mount it onto the container at `/tmp`, using the bind path option in the container.

- If local storage does not exist, use NFS or Lustre and mount it at `/tmp` in the container.

2.4 About Dask

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

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

Dask on Urika®-CS 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®-CS uses to coordinate workers with the Client and Scheduler.

CAUTION: Dask Distributed version 1.20 is not compatible with Urika-CS. 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 , therefore, it is recommended to use newer Dask Distributed versions.

For more information, refer to *Use Dask to Run Python Programs* on page 11

Urika®-CS 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.

2.5 About Intel BigDL

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

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

For more information, visit *https://bigdl-project.github.io/0.5.0/* 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®-CS

The version of BigDL used on Urika®-CS is 0.5.0. BigDL is built with MKL support and is pre-installed on Urika®-CS. The BigDL distribution package is located under `/opt/bigdl-0.5.0/dist` in the Urika®-CS 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.

3 Perform Machine Learning Tasks

3.1 Execute Commands Inside Containers Using the `run_training` Script

The `run_training` script executes commands inside a Singluarity 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.

3.2 Use Dask to Run Python Programs

About this task

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-CS `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-CS. 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
$ module load openmpi/gcc/64/3.0.0
$ start_analytics -k -e mydaskenv
Analytics cluster ready.  Type 'spark-shell' for an interactive Spark shell.
(mydaskenv)
```

The path shown in the preceding example for loading the openMPI module depends on the system.

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'])
>>>
```

3.3 Start an Analytics Cluster and Run OSA Jobs Using the `start_analytics` Command

The `start_analytics` command starts an analytics cluster, which can be used to run Open Source Analytics (OSA) components, including Spark, Anaconda, Dask, BigDL, TensorFlow, TensorBoard and Jupyter Notebook. It can be considered as an entry point to the OSA components. The `start_analytics` command 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.

3.4 Start Up an Analytics Cluster and the Analytic Programming Environment

Prerequisites

This procedure assumes that the workload manager being used is either Slurm or PBS Pro. Contact a Cray service representative if a different workload manager is being used.

About this task

Procedure

1. Load the `analytics` module.

   ```
   $ module load analytics
   ```

2. Optional: Set values for environment variables if needed. For more information, refer to *Start an Analytics Cluster and Run OSA Jobs Using the start_analytics Command* on page 12

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

   Example for Slurm:

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

   Example for PBS Pro:

   ```
   $ qsub -I -lnodes=numberofNodes
   $ module load analytics
   $ module load openmpi/gcc/64/3.0.0
   $ start_analytics
   ```

   The path shown in the preceding example for loading the openMPI module depends on the system.

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

3.5 Enable Anaconda Python and the Conda Environment Manager

Prerequisites

This procedure assumes that Slurm or PBS Pro is being used as the workload manager. Contact Cray support if using other workload managers.

About this task

Urika-CS 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-CS. 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
$ module load openmpi/gcc/64/3.0.0
```

The path shown in the preceding example for loading the openMPI module depends on the system.

3. Start an analytics cluster.

```
$ start_analytics
```

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

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

4. Create a Conda environment.

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

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

IMPORTANT: Use the conda `config --add envs_dirs` *`path_to_directory`* command if it is required to set an alternate environments directory for Conda. *`path_to_directory`* must be a directory that is mounted within the container. This is particularly useful when the home directory space is limited.

5. Activate the Conda environment.

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

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

3.6 Create New Conda Environments with TensorFlow

Prerequisites

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

About this task

By default, two TensorFlow libraries of versions 1.6.0 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-CS 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 openmpi/gcc/64/3.0.0
   $ module load analytics
   $ start_analytics
   ```

   The path shown in the preceding example for loading the openMPI module depends on the system.

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-1.6.0-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
```

3.7 Visualize Statistics with TensorBoard

About this task

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

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

Procedure

1. 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
$ module load openmpi/gcc/64/3.0.0
```

   The path shown in the preceding example for loading the openMPI module depends on the system.

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.

3.8 Get Started with Intel BigDL

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

- Run `spark-shell` with BigDL.

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

- Use the BigDL Tensor API.

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

- Use the LeNet on MNIST "Hello World" deep learning example, which trains LeNet-5 on the MNIST data using BigDL. For more information, visit *https://bigdl-project.github.io/0.5.0/* 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/0.5.0/*. These examples are pre-built with the BigDL distribution and demonstrate how to use BigDL to train and evaluate several of the supported neural network models. Use the following bash script to call one of these pre-built examples:

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

  $ spark-submit --total-executor-cores $cores \
  --conf spark.executor.instances=$worker_nodes --conf spark.executor.cores=20 \
  --conf spark.shuffle.reduceLocality.enabled=false \
  --class com.intel.analytics.bigdl.models.lenet.Train \
  $BIGDL_DIR/lib/bigdl-0.5.0-jar-with-dependencies.jar \
  -f /lus/snx11254/userName/mnist -b $batch_size -r 0.10 \
  --checkpoint ./tests/log/model # echo "Exiting function: launchBigDLJob"
  }
  ```

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

Prerequisites

This procedure assumes that the workload manager being used is either 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/0.5.0/* 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
   $ module load openmpi/gcc/64/3.0.0
   $ start_analytics
   ```

   The path shown in the preceding example for loading the openMPI module depends on the system.

   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-0.5.0-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.

3.10 Run Intel BigDL Programs Using PySpark

Prerequisites

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

About this task

This procedure enables users to run PySpark applications on Urika®-CS 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
   ```

```
$ module load openmpi/gcc/64/3.0.0
$ start_analytics
```

The path shown in the preceding example for loading the openMPI module depends on the system.

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-0.5.0-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 preceding, `-b`: Specifies the mini-batch size. It is expected that the mini-batch size is a multiple of *`node_number`* * *`core_number`*, i.e., the product of the number of nodes and the number of cores-per-node

   ```
   $ spark-submit --total-executor-cores 640 --conf spark.executor.instances=32 \
   --conf spark.executor.cores=20 --py-files ${PYTHON_API_ZIP_PATH},\
   ./tests/py_files/v0.5.0_py3/textclassifier.py --jars ${BIGDL_JAR} \
   --conf spark.executorEnv.PYTHONHASHSEED=123 \
   ./tests/py_files/v0.5.0_py3/textclassifier.py -b 2560 --max_epoch 3 --model cnn
   ```

3.11 Run Intel BigDL Programs as Local Java or Scala Programs

Prerequisites

This procedure assumes that the workload manager being used is either 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
   ```

   If using a PBS Pro based system, also load the openMPI module as show in the following example:

   ```
   $ module load openmpi/gcc/64/3.0.0
   ```

   The path shown in the preceding example for loading the openMPI module depends on the system.

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.

3.12 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
$ module load openmpi/gcc/64/3.0.0
```

The path shown in the preceding example for loading the openMPI module depends on the system.

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 -N -f -L localPort:localhost:loginPort CS_STORM_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.

3.13 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.

   `CS_STORM_HOSTNAME` is used as an example for the login node's name in the following example:

   ```
   $ ssh CS_System_HostName
   ```

2. Load the `analytics` module

   ```
   $ module load analytics
   ```

3. Obtain a job allocation.

   Example for Slurm:

```
$ salloc -N numberofNodes
```

Example for PBS Pro:

```
$ qsub -I -lnodes=numberofNodes
$ module load analytics
$ module load openmpi/gcc/64/3.0.0
```

The path shown in the preceding example for loading the openMPI module depends on the system.

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

```
$ ssh -N -f -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.

Example for Slurm:

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

Example for PBS Pro:

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

Here:

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

6. Start the Jupyter Notebook application.

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

```
$ jupyter notebook --port UIPort --notebook-dir=./ --no-browser
[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.

4 Cray PE ML Plugin - Cray Distributed Training Framework

Deep Learning (DL) with neural networks can be used as a critical tool for academia because of its transformative potential for a wide variety of problems. The amount of computational resources needed to train sufficiently complex networks can limit the use of DL, however. High Performance Computing (HPC), in particular efficient scaling to large numbers of nodes, is ideal for addressing this problem. This section describes the Cray PE ML plugin, a portable distributed training framework for scaling deep learning training workloads.

Distributed Deep Learning

Stochastic gradient descent (SGD) is the optimization technique most commonly used to train deep neural networks. The training process requires a large training dataset labeled with information about each sample that the network needs to learn. One of the SGD steps uses a random subset of that data, called the mini-batch, to compute partial derivatives for each tunable parameter in the network. These derivatives/gradients measure the difference between the output of the network and the correct result given by the labels. Each sample in the random subset produces its own gradients. The gradients from each sample are averaged together, after which the average is used to update the network parameters for the next SGD step. Modifications to SGD typically involve new optimizers. This is the method used to update the model given a set of gradients.

SGD can be parallelized by subdividing a sufficiently large mini-batch evenly among a set of processes. Each process computes gradients locally, and then communicates them to produce globally averaged gradients. The neural network parameters (model) are then updated with these gradients. This technique is called synchronous data parallel SGD (SSGD).

It is possible to reduce DL training time to accuracy with SSGD by increasing the global mini-batch size (sum across all processes) and increasing the SGD step size, also referred to as the learning rate. The errors on globally averaged gradients decrease as more samples are used to compute them. Lower errors allow for larger updates to the model at each step, potentially leading to faster convergence. There are limitations to how far the global mini-batch size can be increased before slow convergence or lack of convergence is observed. More sophisticated optimizers must then be used to overcome these training difficulties. Adaptive optimizers such as Adam and LARS are proven examples for certain classes of networks.

Distributed TensorFlow

Many parallelization frameworks for DL, such as gRPC in TensorFlow, consist of two classes of processes: Parameter Servers (PS) and workers.

Parameter Server (PS) processes gather gradients from worker processes, compute the globally averaged gradients, update the network parameters, and send the new parameter values to workers. Typically the user can select the number of PS processes. A single or limited number of PS processes with a large number of workers will encounter performance issues and limited scaling. This configuration establishes a many-to-few communication pattern, which causes congestion on most networks. It is also difficult for a limited number of PS processes to send out updated parameter values fast enough to keep up with worker demand. Increasing the number of PS processes can reduce the communication and parameter update bottlenecks. Using too many PS processes, however, results in many-to-many communication patterns, which will not scale to large numbers of nodes. Determining the optimal number of PS processes is very cumbersome for users. For gRPC in TensorFlow, users must also provide node names and port numbers, adding to usability issues.

Cray PE ML Plugin – Cray Distributed Training Framework

The Cray PE ML plugin addresses the scaling performance and usability issues in TensorFlow and other similar DL frameworks. There are no PS processes when using the Cray PE ML plugin. Every process is a worker, and a custom global reduction operation replaces the gradient aggregation work of PS processes. Every worker then redundantly computes network parameter updates, which is typically a small fraction of the execution time. The Cray PE ML plugin has a custom reduction for gradient aggregation, specifically optimized for DL workloads. In addition to improved communication performance at scale, the custom reduction implementation also offers excellent computation/communication overlap. The ability to hide the communication costs of the average gradient computation phase plays a key role in improving the time to accuracy of distributed training.

No modifications to TensorFlow are required to use the Cray PE ML plugin for parallelization. The TensorFlow custom Op feature is used to add the necessary communication steps to the execution graph in an optimal way. Users can start with a serial TensorFlow or other framework client script and add the required calls for initialization, communication, and finalization. For situations where multiple gradient reductions are needed, such as GANs, teams of reduction threads can be used to accelerate each independently with a few simple calls. C/C++ and Python 2 or Python 3 interfaces are available with the Cray PE ML plugin.

High Performance Parallel SGD Algorithm

The Cray PE ML plugin has advanced parallelization algorithm features that enable even higher parallel efficiency over that provided for pure SSGD. This advanced algorithm uses non-blocking communication and pipelining to hide communication behind gradient computation with exceptional efficiency. The primary algorithm, referred to as Delayed Synchronous SGD (DSSGD) has two paths to select from; warm-up and cool-down. The warm-up path is preferred for simple optimizers, such as SGD or momentum SGD. The cool-down path is preferred for sophisticated optimizers like Adam. DSSGD is described by the following psuedo-code, where `M_step` are the model values at the current step:

DSSGD Algorithm with Warm-up

```
for 0 <= step < K do
    B_local,step = local mini-batch
    G_local,step = compute_gradients(M_step, B_local,step)
    G_global,step = allreduce_blocking(G_local,step)
    M_step+1 = apply_gradients(G_global,step)

    if step = K-1 then
        start_allreduce_nonblocking(G_local,step)
    end if
end for

for K <= step < max_steps do
    B_local,step = local mini-batch
    G_local,step = compute_gradients(M_step, B_local,step)
    G_global,step = end_allreduce_nonblocking()

    if step < max_steps-1 then
        start_allreduce_nonblocking(G_local,step)
    end if
    M_step+1 = apply_gradients(G_global,step)
end for
```

DSSGD Algorithm with Cool-down

```
for 0 <= step < K do
    B_local,step = local mini-batch
    G_local,step = compute_gradients(M_step, B_local,step)

    if step > 0 then
        G_global,step = end_allreduce_nonblocking()
    else
        G_global,step = allreduce_blocking(G_local,step)
    end if

    if step < max_steps-1 then
        start_allreduce_nonblocking(G_local,step)
    end if

    M_step+1 = apply_gradients(G_global,step)
end for

for K <= step < max_steps do
    B_local,step = local mini-batch
    G_local,step = compute_gradients(M_step, B_local,step)

    if step = K then
        G_global,step = end_allreduce_nonblocking()
    else
        G_global,step = allreduce_blocking(G_local,step)
    end if

    M_step+1 = apply_gradients(G_global,step)
end for
```

DSSGD consist of two phases:

- Blocking communication phase - The blocking communication phase exactly replicates pure SSGD and has identical convergence behavior.
- Non-blocking communication phase - The non-blocking communication phase is a unique form of asynchronous SGD where every process has identical models (as in SSGD) but gradients are derived from model parameters one step out of date. This has a slight drag on convergence, but when properly tuned these algorithms can converge in close to the same number of mini-batch steps as SSGD but with even higher parallel efficiency.

In addition to the algorithm selection described earlier, users must tune the value of parameter K, which defines the mini-batch step where a transition between blocking and non-blocking communication phases occurs. The non-blocking communication phase can be used for approximately 10-90% of the training run depending on the exact nature of the model being trained and optimizer in use. Users should start with K set to the total number of mini-batch steps to train to tune optimizers and hyper-parameters at the chosen job size. This forces pure SSGD to be used for the duration of training. Once other hyper-parameters are set, a coarse bi-section search of K can be used to find the optimal setting for both performance and accuracy.

4.1 Port Scripts to Use the Cray Programming Environment Machine Learning Plugin

Prerequisites

This procedure requires Urika-CS software installed on a Cray CS system.

About this task

To port a TensorFlow training script to use the Cray PE ML plugin, it is recommended to start with a training that executes serially, i.e., it does not use distributed TensorFlow. With that script, the following modifications are necessary to use the plugin:

1. Initializing the Cray PE ML plugin, specifying the number of teams, threads, model size
2. Broadcasting initial model parameter values
3. Using the Cray PE ML plugin to communicate gradients after gradient computation and before the model update
4. Finalizing the Cray PE ML plugin

Modifications that are not required by the Cray PE ML plugin, yet common for parallelization, include:

- Correcting the definition of an epoch for the global mini-batch size (all processes)
- Correcting the learning rate:
  - Linear or square root scaling rule
  - Adding a learning rate decay schedule
- Average performance metrics using Cray PE ML plugin helper functions

  Only a single rank produces the desired prints. In addition, either a single rank writes checkpoints or each rank writes to a unique location

The following code excerpts (from the MNIST example provided with the release) review the required modifications for using the Cray PE ML plugin.

Procedure

1. Initialize the Cray PE ML plugin.

   This is done by first importing the `ml_comm` module and calling its `init()` function as follows:

   Example of initializing the Cray PE ML plugin

```
# import the Cray PE ML Plugin module
import ml_comm as mc

def main():

  # Build the model
  model=build_model(n_in, n_layer, n_hid, n_out)

  # if the Plugin is enabled initialize it
  if (flags.enable_ml_comm == 1):
      # determine the model size
      totsize = sum([v for v in tf.trainable_variables()])

      # initialize the Cray PE ML Plugin
      mc.init(nthread_per_team=2, nteams=1, msglen=totsize, "tensorflow")
      ...
```

Note the call to `mc.init()` includes the additional argument of `tensorflow`, which indicates that TensorFlow is the training framework. There are two effects from specifying `tensorflow`. First, calls to the Cray PE ML plugin operations for broadcasting initial model parameters (`mc.broadcast()`) and gradient aggregation (`mc.gradients()`) switch from expecting NumPy array data buffers to native TensorFlow Tensor buffers. Second, TensorFlow operations for both broadcast and gradient aggregation get loaded, which allows those calls to be added to the execution graph.

The next initialization step is to configure the single thread team to be aware of the total number of training steps. Users first decide on the number of training epochs and number of samples in a mini-batch per process. The total number of training steps M_{steps} is then given by:

$$M_{steps}=n_{train}/(k_{ranks}\times b_{local})\times N_{epochs}$$

where N_{epochs} is the number of training epochs, n_{train} is the number of samples in training dataset, k_{ranks} is the number of workers (MPI ranks or processes), and b_{local} is the local batch size.

The configuration step is shown below with multiples suggestions about how to modify the configuration, continuing from the previous code block:

Example of configuration the Cray PE ML plugin initialization parameters

```
...
# use the CPE ML Plugin to get our rank and
# the number of processes
myrank     = mc.get_rank()
numworkers = mc.get_nranks()

# num_train_samps is the number of samples in
# our training data set
max_steps = int(math.ceil(flags.train_epochs * (num_train_samps) / (numworkers
* batch_size)))

# configure the single thread team and have rank 0 print out communication
performance metrics
# every 100 steps
mc.config_team(0, 0, max_steps, max_steps, 1, 100)


# The above configuration can be modified as needed. For example, to configure
the team to complete
# a blocking warm-up phase for the first 10% of training without a smooth
transition:
# mc.config_team(0, 0, int(0.10 * max_steps), max_steps, 1, 100)
#
# To instead use a smooth transition:
# mc.config_team(0, 1, int(0.10 * max_steps), max_steps, 1, 100)
#
# To perform a cool-down blocking phase for the final 10% of training, make
the following change:
```

```
# mc.config_team(0, 1, -1*int(0.90 * max_steps), max_steps, 1, 100)
...
```

2. Broadcast initial model parameters.

With Tensorflow, it is ideal to use a SessionRunHook to manage the broadcast and any desired averaging of metrics. Session hooks can be passed to convenience classes, such as Estimator, to be run at specific points in session management. A typical SessionRunHook for broadcasting initial model parameters is defined as follows:

Example of broadcasting initial parameters in the SessionRunHook class

```
class BcastTensors(tf.train.SessionRunHook):

  def __init__(self):
    self.bcast = None

  def begin(self):
    if not self.bcast:
      new_vars   = mc.broadcast(tf.trainable_variables(),0)
      self.bcast = tf.group(*[tf.assign(v,new_vars[k]) for k,v in
enumerate(tf.trainable_variables())])

  def after_create_session(self, session, coord):
     session.run(self.bcast)
```

This `SessionRunHook` broadcasts the model parameters from rank 0 to all other MPI ranks and then assigns the new values. The actual operation is performed after the session is created. This `SessionRunHook` must be provided to the Estimator instance as follows:

Example of supplying SessionRunHook to the Estimator instance

```
train_hooks = None

# if the Cray PE ML Plugin is enabled add the hook
if (mlcomm == 1):
    train_hooks = [BcastTensors()]

cnf = tf.estimator.EstimatorSpec(mode=mode,
                                 predictions=predictions,
                                 loss=loss,
                                 train_op=train_op,
                                 training_hooks=train_hooks,
                                 eval_metric_ops=metrics)

classifier = tf.estimator.Estimator(model_fn=cnf)

classifier.train(input_fn=input_fn_train, steps=tsteps,
max_steps=flags.max_train_steps)
```

3. Perform gradient aggregation.

The communication and performance intensive operation that is highly optimized in the Cray PE ML plugin is gradient aggregation. This is placed between gradient computation and model update as follows:

Example of gradient aggregation and communication

```
def train(model_function, train_samp, eval_samp,
        batch_size)
```

```
    ....
    # often a serial code will use the
    # optimizer minimize() method
    if (mlcomm != 1):

      minimize_op = optimizer.minimize(loss, global_step)

    else:
      # for the Cray PE ML Plugin
      # we need to split out the minimize call below
      # so we can communicate/average gradients
      grads_and_vars = optimizer.compute_gradients(loss)

      grads     =  mc.gradients([gv[0] for gv in grads_and_vars], 0)
      gs_and_vs = [(g,v) for (_,v), g in zip(grads_and_vars, grads)]

      minimize_op = optimizer.apply_gradients(gs_and_vs, global_step)
    ...
```

It is common for a serial training script to use the `minimize()` method of an optimizer. This method computes gradients and updates the model with those gradients. The global reduction of local gradients must be done between those steps for data parallel training, however. In the code block above, `minimize()` is split into `compute_gradients()` and `apply_gradients()` so that the `mc.gradients()` call can be added. This operation is added to the execution graph and will have direct access to gradient Tensors, located in CPU or GPU memory, without additional buffering.

4. Finalize the plugin.

The final required step for porting a serial training script is to finalize the Cray PE ML plugin, similar to finalizing MPI. This call should be added after training is complete and the session is closed. Often this is at the end of the `main()` function.

Example of finalizating the Cray PE ML plugin

```
def main():

    ...
    # training is complete and we're ready to exit
    mc.finalize()
```

Cray provides several examples with the Cray PE ML plugin package for users to reference. MNIST and `tf_cnn_benchmarks` examples are provided as part of this release. The `tf_cnn_benchmarks` example is commonly used to benchmark the performance across a set of standard CNNs.

4.2 Run TensorFlow Inside Urika-CS Containers Using the Cray PE ML Plugin

Prerequisites

This procedure requires:

- Urika®-CS 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 PE 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®-CS start with a serial (non-distributed) Python training script, include a few simple lines for the Cray PE 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 Cray PE ML plugin to obtain better performance by by-passing gRPC setup. The Cray PE ML plugin has both C and Python interfaces for the communication needs of DL training.

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-py3/1.1.1/examples/tf_mnist/mnist.py`. The script is documented with any modifications, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.1.1/examples/tf_mnist/README` also describes the modifications.
- **`tf_cnn_benchmarks`** - This is an example of modifying a script already able to run across multiple nodes through gRPC to instead use the CPE ML plugin. Both capabilities (gRPC and the Cray PE ML 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 Cray PE ML plugin. The source files for this benchmark are located in: `/opt/cray/pe/craype-ml-plugin-py3/1.1.1/examples/tf_cnn_benchmarks`. Any modifications are documented inside the source files, and the file `/opt/cray/pe/craype-ml-plugin-py3/1.1.1/examples/tf_cnn_benchmarks/README` describes the changes in detail.

Tuning Recommendations for CPU and GPU Nodes

For CPU and single GPU nodes, the best performance is typically achieved with one MPI rank per node. The Cray PE ML plugin should be configured to use 2-4 communication threads with the `mc.config_team()` interface. In some cases with GPU nodes, performance can be improved using up to 8 threads. For training with MKL and MKL-DNN, it is important to not set `OMP_NUM_THREADS` too high, lest cores become oversubscribed. For example, if there are 36 physical cores on a node, optimal performance is achieved with `OMP_NUM_THREADS=34` while leaving two cores/threads for communication with the Cray PE ML plugin. Additionally, with TensorFlow and the tf_cnn_benchmarks example, `num_intra_threads` should be set to match the value of `OMP_NUM_THREADS`, and `num_inter_threads` can typically be set to 1-3 threads depending on the number of HyperThreads available per core. For KNL CPUs, it is typically best to leave one HyperThread free on each core. The `KMP_BLOCKTIME` environment variable may yield slightly improved performance if set to 0 or 30.

For GPU nodes, the number of CUDA streams used to buffer data to the host can be modified via the `ML_COMM_NUM_CUDA_STREAMS` environment variable, and the number of copies each of those streams performs is changed with the `ML_COMM_CPY_PER_CUDA_STREAM` environment variable. The default settings, 2 and 8, respectively, have empirically been found to be best for nearly all situations.

When executing on multi-GPU nodes such as CS Storm 500GT or CS Storm 500NX nodes, best performance may be achieved in different manners for single node and multi-node execution. For single node execution, the

Cray PE ML plugin typically runs best with 1 rank per GPU, i.e.: 8 ranks on the node. In that case, two communication threads provide good performance. With multi-node execution and depending on the model being trained, running 2 ranks per node, i.e.: 4 GPUs per rank, may be more optimal, while continuing to use 2 communication threads per rank.

This also requires changes to be made to the TensorFlow training script such that each MPI rank only sees the number of GPUs it is allowed to use. This is handled via TensorFlow's `tf.ConfigProto()`:

Masking GPUs from TensorFlow

```
# To execute a multi-node job, where two ranks execute on a node and each use 4
GPUs
config = tf.ConfigProto()
if (mc.get_rank %2 == 0):
  config.gpu_options.visible_device_list = '0,1,2,3'
else:
  config.gpu_options.visible_device_list = '4,5,6,7'


# If running a single node training job, you may want to launch 8 MPI ranks with
each using a GPU. In that case,
# the visible device list could be changed like so:
# config.gpu_options.visible_device_list = str(mc.get_rank())
```

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. Load the OpenMPI module. Check the installation instructions for details on OpenMPI module file.

```
$ module load openmpi/gcc/3.0.0
```

3. Allocate the desired number of nodes in interactive mode or as part of a SLURM or PBS job submission script.

If the CS 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
```

4. Switch to the current working directory to copy the contents of `/opt/cray/pe/craype-ml-plugin-py3/1.1.1/examples/tf_cnn_benchmarks/*` (which are the TensorFlow examples packaged with the plug-in) to the current working directory if it is required to run the `tf_cnn_benchmark` example provided with the CPE ML plug-in.

```
$ cd workingDir
$ cp -r /opt/cray/pe/craype-ml-plugin-py3/1.1.1/examples/tf_cnn_benchmarks/* .
```

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

   Submit a TensorFlow command to the `run_training` command. 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 v7.1.2 library located at `/home/users/user/CuDNN/cudnn-9.0-v7.1.2/cuda/lib64`

```
$ run_training -n 2 --ppn 1 --craype-plugin --cudnn-libs \
/home/users/user/CuDNN/cudnn-9.0-v712/cuda/lib64 \
--no-node-list "python tf_cnn_benchmarks.py --num_gpus=1 \
--batch_size=64 --model=inception3  --train_dir=/home/users/user/tf_cnn_train \
--data_name=imagenet --variable_update=ps_ml_comm  --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*.

   Intel Xeon example for Broadwell dual socket 18 core nodes:

```
$ $ run_training -n 2 --ppn 1 --craype-plugin \
--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:

   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 --craype-plugin --cudnn-libs /home/users/username/CuDNN/cudnn-9.0-v712/cuda/lib64
```

   To run the plugin on a single node using interactive shell on a CPU node:

```
$ salloc -N 2 -p bdw
$ start_analytics --craype-plugin
```

   This starts an interactive session inside the container.

```
prod-001 $ export PYTHONPATH=/opt/tensorflow_cpu:$PYTHONPATH
prod-001 $ 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
```

   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)
```

5 Urika-CS Troubleshooting Information

Module Considerations

- If a module is not available and module files are not recognized by the system, skip the `module load analytics` command in the following sections.
- If a module is not available and Urika-CS is already installed on the system, set the PATH to point to Urika-CS installation directory as described in *Install Urika-CS Software*.

Consideration for Running on PBS Pro Based Systems

On PBS Pro run the following commands after allocating and loading the required modules.

For example:

```
$ qsub –I –lnodes=4
$ module load analytics
$ module load openmpi/gcc/64/3.0.0
```

The path shown in the preceding example for loading the openMPI module depends on the system.

After executing the preceding commands, execute `start_analytics` or `run_training`, specifying the required options.

5.1 Resolve I/O Related Issues while Using the Cray PE ML Plugin

Users running trainings on a large number of nodes should consider the associated potential I/O bottlenecks. TensorFlow provides the `Dataset` API, which has features for dedicating threads to reading training samples from disk asynchronously behind gradient computation. Users are strongly encouraged to use the `Dataset` API for best I/O performance. However, at large node counts, subsystems may not be able to deliver the required read bandwidth to prevent sample starvation for workers. The issue may at first appear as limited scaling efficiency but not be due to communication performance. Users are encouraged to include a method for training with dummy data in their training script, so that it does not require any I/O. This capability can be used to identify if I/O is a performance bottleneck.

5.2 Resolve Convergence Related Issues Encountered while Using the Cray PE ML Plugin

About this task

Data parallel training with very large global mini-batch sizes can result in slow convergence or lack of convergence without additional tuning of hyperparameters or the selected optimizer. It is recommended that users observing poor convergence try out the following procedure to improve performance:

Procedure

1. Configure the Cray PE ML plugin to use pure SSGD by setting the `K` parameter to `mc.config_team()` to the maximum number of training steps
2. Increase the learning rate following either the `sqrt()` or linear scaling rules. The `sqrt()` rule states that for an increase of *N* workers, the learning rate should increase by `sqrt(N)`. The linear rule states that for an increase of *N* workers, the learning rate should increase by *N*. The linear rule is preferred for faster convergence but potentially less stable.
3. Add a learning rate decay schedule to reduce the learning rate from a large initial value based on step (2) to a small value as training finds a desirable local minimum. Examples are a linear decay or polynomial decay both supported by TensorFlow.

   More sophisticated optimizers are required for extremely large global mini-batch sizes. The exact mini-batch size where this occurs is model and dataset dependent. Examples of sophisticated optimizers are Adam, LARS, and LARC with Adam as the base optimizer. Users may have to implement such optimizers manually in TensorFlow.

5.3 Troubleshoot NVIDIA, CUDA and CuDNN Related Issues

Execute the following commands on GPU nodes to verify the Singularity configuration settings with respect to GPU libraries:

```
$ module load analytics
$ salloc -N 2
$ run_training -n 2 --cudnn-libs path_to_cudnn "python -c 'import tensorflow as tf; print(tf.__version__)'"
1.6.0
/opt/anaconda/lib/python3.6/site-packages/h5py/__init__.py:34: FutureWarning: \
Conversion of the second argument of issubdtype from `float` to `np.floating` \
is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

Use the following table to resolve issues encountered while executing the preceding command:

Table 2. NVIDIA, CUDA and CuDNN Related Issues and Resolutions

Error Message	Resolution
ImportError: libcublas.so.9.0: cannot open shared object file: No such file or directory	Check if CUDA toolkit is mounted per the installation instructions.

Error Message	Resolution
libcuda.so.1: cannot open shared object file: No such file or directory	Ensure that the Nvidia drivers for `GPU - libcuda.so.1` are located under `/usr/lib64` on the compute nodes. If this is not the location, pass the `--gpu-libs` as an additional argument to the `run_training` command
ibcudnn.so.7: cannot open shared object file: No such file or directory	Ensure that CuDNN version used is v7.1.2 for CUDA toolkit 9.0.

For more information, refer to installation of Nvidia, CUDA, and CUDA toolkit in *Install Supporting Software*

5.4 Troubleshoot Cray PE ML Plugin and TensorFlow Related Issues on ACE Based Systems

Execute the following commands to verify that functionality of the Cray PE ML plugin with TensorFlow.

```
$ module load analytics
$ module load openmpi/gcc/3.0.0
$ salloc -N 2
$ run_training -n 2 --craype-plugin --no-node-list \
'python $CRAYPE_ML_PLUGIN_BASEDIR/examples/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
--num_gpus=1 --data_format=NHWC --batch_size=32 --model=trivial --train_dir=$HOME/tf_cnn_train \
--data_name=imagenet --variable_update=ps_ml_comm  --local_parameter_device=cpu'

Step    Img/sec loss
1   images/sec: 3.2 +/- 0.0 (jitter = 0.0)  10.681
1   images/sec: 3.3 +/- 0.0 (jitter = 0.0)  10.681
```

If there are issues running TensorFlow using the Cray PE ML plugin, check if `openmpi` has been installed. If the module file already includes settings for the `OPAL_PREFIX` and `OMPI_DIR` variables, setting them again is not required.

Use the following table to troubleshoot any errors returned by executing the preceding commands:

Table 3. Cray PE ML Plugin and TensorFlow Related Issues and Resolutions

Error	Resolution
• Sorry! You were supposed to get help about: opal_init:startup:internal-failure But I couldn't open the help file: /global/opt/ompi/share/openmpi/help-opal-runtime.txt: No such file or directory. Sorry! • Sorry! You were supposed to get help about: orte_init:startup:internal-failure But I couldn't open the help file: /global/opt/ompi/share/openmpi/help-orte-runtime: No such file or directory. Sorry! • Sorry! You were supposed to get help about: mpi_init:startup:internal-failure But	Mount the OpenMPI installation if it is not installed in a standard location. For example in the first example on the left, use the `--mount` option with the `run_training` command, as shown in the following example: `--mount /gloabal/opt/ompi:/global/opt/ompi` Alternatively, add it to the Singularity configuration as the bind path.

Error	Resolution
I couldn't open the help file: /global/opt/ompi/share/openmpi/help-mpi-runtime.txt: No such file or directory. Sorry! ● Sorry! You were supposed to get help about: mpi_init:startup:internal-failure But I couldn't open the help file: /global/opt/ompi/share/openmpi/help-mpi-runtime.txt: No such file or directory. Sorry! ● *** An error occurred in MPI_Init_thread *** on a NULL communicator *** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abort, *** and potentially your MPI job) [prod-0077:80273] Local abort before MPI_INIT completed completed successfully, but am not able to aggregate error messages, and not able to guarantee that all other processes were killed!	
We encountered issues with Slurm and OMPI inter-dependency.	Ensure that OpenMPI dependent libraries are available at `/usr/lib64`. In addition, ensure that this directory is mounted in the Singularity configuration at `/opt/hostlib`

ibibverbs: Warning: couldn't load driver 'librxe-rdmav2.so': \ librxe-rdmav2.so: cannot open shared object file: No such file or directory libibverbs: Warning: couldn't load driver 'libqedr-rdmav2.so': \ libqedr-rdmav2.so: cannot open shared object file: No such file or directory mca_base_component_repository_open: unable to open mca_btl_openib: \librdmacm.so.1: cannot open shared object file: No such file or directory	Ensure that the `libibverbs` dependent libraries are available at `/usr/lib64` on all the nodes.
	● On Slurm based systems, execute:
	`$ ompi_info`
	`Configure command line: \`
	`'--prefix=/global/opt/openmpi/3.0.0/gcc'`
	`'--with-pmi=/opt/local/slurm/default'`
	● On PBS Pro based systems, execute:

For more information, refer to installation of OpenMPI in *Install Supporting Software*

5.5 Troubleshoot Apache Spark Related Issues

Execute the following command to verify that Spark is working as expected:

```
$ module load analytics
$ salloc -N 2
$ start_analytics
>> run-example SparkPi
```

- Example result of a successful execution:

```
Pi is roughly 3.1433757168785843
```

- Example result of failure:

```
/opt/rt_scripts/bin/setup_rt_env.sh: line 347: /root/.urikacs/startuplogs/job.
26/
master.log: No such file or directory
Running the 2 worker images (/opt/cray/analytics-singularity-images/
1.01.0000.201806061400_0108/analytics-urikacs-1.0.0000-latest.simg).
Logging to /root/.urikacs/startuplogs/job.26/workers.log. \
/opt/rt_scripts/bin/setup_rt_env.sh: line 347: \
/root/.urikacs/startuplogs/job.26/ workers.log: \
No such file or directory
/opt/rt_scripts/bin/setup_rt_env.sh: \
line 347: /root/.urikacs/startuplogs/job.26/ workers.log:\
No such file or directory
..... .
18/06/08 20:21:08 ERROR \
SparkContext: Error initializing SparkContext.
org.apache.spark.SparkException: \
Invalid master URL: spark://:7077
The preceding result indicates that the directory /root/.urikacs is not
writable from within the container. \
Rerun the command as a non-root user to resolve the issue
```

The preceding result indicates that the directory `/root/.urikacs` is not writable from within the container. Rerun the command as a non-root user to resolve the issue.

5.6 Troubleshoot gPRC and TensorFlow Related Issues

Execute the following command to verify that gPRC and TensorFlow are working as expected:

```
$ module load analytics
$ salloc -N 2
$ run_training -n 2 --no-node-list 'python $CRAYPE_ML_PLUGIN_BASEDIR/examples/tf_cnn_benchmarks/tf_cnn_benchmarks.py \
--num_gpus=1 --batch_size=32 --model=trivial --variable_update=parameter_server'

Step    Img/sec loss
1       images/sec: 2.7 +/- 0.0 (jitter = 0.0)  30.208
1       images/sec: 2.7 +/- 0.0 (jitter = 0.0)  30.208
10      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  96.949
10      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  96.949
20      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  147.888
20      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  147.888
30      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  238.592
30      images/sec: 2.7 +/- 0.0 (jitter = 0.0)  238.592
```

The preceding result indicates that GPRC and TensorFlow are working as expected. However, if the preceding command returns an error, verify that the Singularity configuration and Cray PE ML plugin directory are mounted properly, as specified in the configuration.

5.7 Troubleshoot Issues Related to Accessibility and User Commands

- **Accessibility Checks**

Ensure that Urika-CS is accessible from all the nodes that it needs to be used on.

```
$ module load analytics
```

If executing the preceding command indicates that the `analytics` module is not available, check if the module has been installed and accessible from that node.

- **User Command Checks**

  Execute the following command to verify that user commands are working as expected:

```
$ module load analytics
$ salloc -N 2
$ run_training -n 2 --no-node-list 'hostname'
prod-0078
prod-0077
$ start_analytics
The default Spark event log directory /lus/scratch/sparkHistory does not exist.
Using /home/users/alice/.urikacs/sparkHistory instead.  To choose an alternate
location, set SPARK_EVENT_DIR.
Using default per-node loopback filesystem for Spark per-node local temporary
storage (controlled by SPARK_LOCAL_DIRS).
WARNING: Too few nodes allocated.  Running a local spark instance instead.
Running the interactive image (/opt/cray/analytics-singularity-images/
1.01.0000.201805291754_0106/analytics-urikacs-1.0.0000-latest.simg).
preparing Spark interactive worker
‘/usr/spark/conf/docker.properties.template’ -> ‘/tmp/alice/spark/conf/
docker.properties.template’
‘/usr/spark/conf/fairscheduler.xml.template’ -> ‘/tmp/alice/spark/conf/
fairscheduler.xml.template’
‘/usr/spark/conf/log4j.properties.template’ -> ‘/tmp/alice/spark/conf/
log4j.properties.template’
‘/usr/spark/conf/metrics.properties.template’ -> ‘/tmp/alice/spark/conf/
metrics.properties.template’
‘/usr/spark/conf/slaves.template’ -> ‘/tmp/alice/spark/conf/slaves.template’
‘/usr/spark/conf/spark-defaults.conf.template’ -> ‘/tmp/alice/spark/conf/spark-
defaults.conf.template’
‘/usr/spark/conf/spark-env.sh.template’ -> ‘/tmp/alice/spark/conf/spark-
env.sh.template’
adding test data to home directory
Analytics cluster ready.  Type 'spark-shell' for an interactive Spark shell.
```

  If the preceding command returns an error message, check if Urika-CS is installed correctly with PATH being set to include Urika-CS installation directory. The Urika-CS installation directory should be available on all the nodes.

5.8 Troubleshoot Singularity Related Issues

- **Installation Checks**

  Use the following commands to check if Singularity has been installed correctly. These commands need to be run on the nodes that Urika-CS needs to be used on.

```
$ salloc -N 2
$ module load singularity
```

Executing the preceding command may cause the system to return the following message if the module file has not have been created in the correct directory.

```
ModuleCmd_Load.c(244):ERROR:105: Unable to locate a modulefile for 'singularity'
```

Ensure that the module file is installed at `/opt/cray/modulefiles/analytics` and then execute the following to ensure Singularity has been installed correctly:

```
$ module load singularity
$ singularity --version
2.5
$ singularity selftest
+ sh -c test -f /global/opt/singularity/2.5.0/etc/singularity/singularity.conf        (retval=0) OK
+ test -u /global/opt/singularity/2.5.0/libexec/singularity/bin/action-suid           (retval=0) OK
+ test -u /global/opt/singularity/2.5.0/libexec/singularity/bin/mount-suid            (retval=0) OK
+ test -u /global/opt/singularity/2.5.0/libexec/singularity/bin/start-suid            (retval=0) OK
```

- **Accessibility Checks**

  Execute the following command to ensure that no error is returned, which indicates that all the compute nodes can access Singularity:

```
$ salloc -N 4
$ module load singularity
$ srun which singularity
```

- **Miscellaneous Checks**

  Check if Singularity can open a shell inside an image:

```
$ module load singularity
$ singularity shell /opt/cray/analytics-singularity-images/default/analytics-urikacs-1.0.0000-latest.simg
```

  Executing the preceding command may cause the system to return one of the following errors:

  - `ERROR : Failed to resolve path to /usr/local/var/singularity/mnt/container`
  - `ABORT : Retval = 255`
  - `ERROR : Failed invoking the NEWUSER namespace runtime: Invalid argument`
  - `ERROR : No valid /bin/sh in container`
  - `ABORT : Retval = 255`

  If any of the preceding errors is returned, check if Singularity has been installed on the NFS mounted drive. Ensure that Singularity is installed on each node and is accessible from all the nodes that Urika-CS needs to run on.

For more information, refer to installation of Singularity in *Install Supporting Software*

5.9 Urika-CS Log File Locations

- Log files for a given Urika®-CS service are located on the node(s) that the respective service is running on.
- 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-CS stores Spark event logs in per-user directories. By default, the location is `/lus/scratch/sparkHistory/` if it is available, or `$HOME/.urikacs/sparkHistory` if it is not. Users 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** - Spark worker logs - These logs reside in the $HOME/.urikacs/sparkHistory directory on the local nodes they run on.

5.10 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:

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.

6 Urika®-CS Quick Reference Information

Major Software Versions

Table 4. Urika®-CS Software Component Versions

Software Component	Version
Apache Spark	2.2.0
Anaconda Distribution of Python	5.0.0
Dask	0.14.3 and later
Dask Distributed	1.16.3 and later
Intel BigDL	0.5.0
Analytics Programming Environment	
Python	3.6 as part of Anaconda 5.0.0. Anaconda also supports creating `python` environments with 2.7, 3.4, and 3.5
Java	1.8
Scala	2.11.8
R	3.5.0
Maven	3.3.9
SBT	0.13.9
ANT	1.9.2
TensorFlow	1.6.0
TensorBoard	1.6.0
Jupyter NoteBook	4.3.0

Urika-CS CLI Commands

Table 5. CLI Command Reference

Command	Description
`start_analytics`	Starts an analytics cluster, which can be used to run Spark and/or the analytic programming environment commands. For more information, refer to the `start_analytics` man page.

Command	Description
`run_training`	Runs a command in a Urika-CS container. For more information, refer to the `run_training` man page.

Environment Variables

Table 6. Environment Variables and Mappings

Environment Variable	Mapping
`ANACONDA_DIR`	`/opt/anaconda`
`JAVA_HOME`	`/usr/lib/jvm/java-1.8.0`
`MAVEN_HOME`	`/usr/share/apache-maven`
`SPARK_VERSION`	`2.2.0`
`SPARK_HADOOP_VERSION`	`2.7`
`SPARK_DIR`	`/usr/spark`
`BIGDL_VERSION`	`0.5.0`
`BIGDL_DIR`	`/opt/bigdl-0.5.0/dist`
`BIGDL_JAR`	`/opt/bigdl-0.5.0/dist/lib/bigdl-0.5.0-jar-with-dependencies.jar`
`SPARK_WORKER_PORT`	`8888`
`DASK_WORKER_PORT`	`19866`
`DASK_NANNY_PORT`	`19868`
`DASK_BOKEH_PORT`	`19870`
`BAZEL_PATH`	`/opt/bazel-0.11.1`
`LD_LIBRARY_PATH`	`/opt/cudnn:/usr/local/lib:/usr/lib/x86_64-linux-gnu:/usr/local/lib:/usr/spark/mathlibs`
`PATH`	`/opt/rt_scripts/bin:/opt/anaconda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`