Best Practice Guide - HPC for Data Science on the Cray Urika

Andreas Vroutsis, EPCC, United Kingdom

Sandra Mendez, LRZ, Germany

Terry Sloan (Editor), EPCC, United Kingdom

Volker Weinberg (Editor), LRZ, Germany

Version 1.0 by 14-01-2019

Table of Contents

1. Introduction 4
1.1. About this document 4
1.2. Guide Structure 4
1.3. Abbreviations and acronyms 4
2. Spark 6
2.1. Introduction 6
2.2. Resilient Distributed Datasets (RDDs) 6
2.2.1. RDD Creation and Transformation 6
2.3. Architecture 7
2.3.1. Spark SQL 7
2.3.2. MLib (Machine Learning) 7
2.3.3. GraphX 8
2.3.4. Spark Streaming 8
2.4. Running Spark on Clusters 8
2.5. Spark Standalone Cluster 9
2.6. Running Spark in Slurm 9
3. Cray Urika GX System 11
3.1. Introduction 11
3.2. Overview 12
3.3. Architecture / Configuration 13
3.3.1. System Level 13
3.3.2. Node Level 13
3.4. System Access 15
3.4.1. Access to the EPCC-hosted Urika 15
3.4.2. User Interfaces 15
3.4.3. Data Transfer (EPCC-hosted Urika) 16
3.5. Production Environment 17
3.5.1. Hadoop and its ecosystem 17
3.5.2. Apache Spark 18
3.5.3. Cray Graph Engine 18
3.5.4. Resource Management 18
3.5.5. File Systems 22
3.5.6. Fault Tolerance 22
3.5.7. User Interfaces 23
3.6. Programming Environment 25
3.6.1. Components 25
3.6.2. Anaconda 25
3.6.3. Jupyter Notebook 25
3.6.4. Apache Spark 26
3.7. Performance Analysis 28
3.8. Tuning 29
3.9. Debugging 30
3.9.1. Tools 30
3.9.2. Log Files 30
Further documentation 32

1. Introduction

1.1. About this document

This best practice guide provides information about exploiting HPC platforms and techniques for Data Science projects.

1.2. Guide Structure

This best practice guide is divided into sections covering specific topics. The contents of each section are briefly described below.

Section	Content
1. Introduction	This section! It describes the guide and its structure.
2. Spark	This is an open source distributed data analytics platform. It provides access to many different data sources and enables parallel computations to be distributed across a cluster. This chapter of the guide describes Resilient Distributed Datasets, the concept at the heart of Spark. It also describes the architecture of Spark, its libraries and how to run Spark on a cluster.
3. Urika GX	The Cray Urika GX system is an HPC platform dedicated to highly interactive and iterative data analytics that require supercomputer levels of computing performance. This chapter describes the production and programming environment on the platform. This environment includes Spark, Hadoop, R and graph databases. The chapter's contents are based on the Urika GX system hosted by EPCC and Cray on behalf of the Alan Turing Institute in the UK.

1.3. Abbreviations and acronyms

Abbreviation	Explanation
API	Application Programming Interface
CGE	Cray Graph Engine
CPU	Central Processing Unit
EPCC	Edinburgh Parallel Computing Centre
GB	GigaByte(s)
HDD	Hard Drisk Drive
HDFS	Hadoop Distributed File System
HDP	Hortonworks Data Platform
HPC	High Performance Computing
I/O	Input/Output
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
ML	Machine Learning
NFS	Network File System

Abbreviation	**Explanation**
PBS	Portable Batch System
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RDF	Resource Description Framework
RPC	Remote Procedure Call
SBT	Scala Build Tool
SCP	Secure Copy Protocol
SFTP	Secure File Transfer Protocol
SQL	Structured Query Language
SSD	Solid State Drive
TB	TeraByte(s)
TCP	Transmission Control Protocol
UAI	Urika-GX Application Interface
UI	User Interface

2. Spark

2.1. Introduction

Apache Spark[3] is a cluster computing framework for large-scale data processing. It is best known for its ability to cache large datasets in memory between jobs. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.

Spark has a large active community. It includes libraries for machine learning, SQL, structured streaming and graph databases. This chapter describes concept at the heart of Spark, namely resilient Distributed Datasets, as well as the architecture of Spark and its libraries. It also explains how to run Spark on a cluster.

2.2. Resilient Distributed Datasets (RDDs)

The RDD concept aims to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient.

An RDD is a distributed collection of data items, for example lines from a text file or sensor data with timestamp and values. An RDD has the following properties:

- Immutability: One can execute an operation on an RDD to produce another RDD but one cannot alter the original RDD.
- Partitioned: An RDD comprises a distributed collection or partitions of items and hence the contents of an RDD can be operated on in parallel. Any operation on an RDD is typically performed using multiple nodes of a computer cluster.
- Resilience: If one of the nodes hosting a partition fails, another of the cluster nodes can takes its data.

Once data is loaded into an RDD, two basic types of operation can be carried out upon it:

- Transformations, which create a new RDD by changing the original through processes such as mapping, filtering, and more;
- Actions, such as counts, which measure but do not change the original data.

The original RDD remains unchanged throughout. The chain of transformations from RDD1 to RDDn are logged, and can be repeated in the event of data loss or the failure of a cluster node.

Transformations are said to be lazily evaluated, meaning that they are not executed until a subsequent action has a need for the result. This will normally improve performance, as it can avoid the need to process data unnecessarily. It can also, in certain circumstances, introduce processing bottlenecks that cause applications to stall while waiting for a processing action to conclude.

Fault-tolerance is achieved, in part, by tracking the sequence of transformations applied to data partitions in an RDD. Efficiency is achieved by parallelization of the data processing across multiple nodes in the cluster, and by minimization of data replication between those nodes.

2.2.1. RDD Creation and Transformation

There are two ways to create RDDs: parallelizing an existing collection, or referencing a suitably formatted dataset in an external storage system (see https://spark.apache.org/docs/latest/rdd-programming-guide.html#external-datasets).

- Parallelizing an existing collection: for example the following Python code calls the Spark sc.parallelize method to create an RDD.

```
data = [1, 2, 3, 4, 5]
```

```
rdd = sc.parallelize(data, 5) # create 5 partitions
```

In this example the number of partitions to create has been set manually to 5 but you can instead get Spark to automatically determine the number of partitions to create based on the size of the cluster.

- Referencing a dataset on distributed storage: for example the following Python code creates a text file RDD using the Spark textfile method.

```
rdd = sc.textFile("data.txt")
```

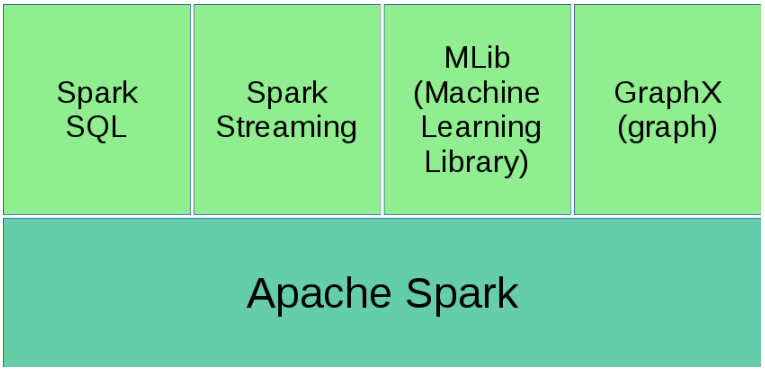
RDDs can be transformed into derived RDDs, for example:

```
rdd2 = rdd.filter( lambda x : (x % 2 == 0) ) # operation: filter odd tuples
```

2.3. Architecture

In addition to RDDs, Spark supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for processing of live data streams (See Figure 1, “Spark Architecture”).

Figure 1. Spark Architecture



2.3.1. Spark SQL

Spark SQL[5] is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API.

To learn more about programming with Spark SQL please refer to the official documentation [5].

2.3.2. MLib (Machine Learning)

MLlib[6] is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering;
- Featurization: feature extraction, transformation, dimensionality reduction, and selection;
- Pipelines: tools for constructing, evaluating, and tuning ML pipelines;
- Persistence: saving and loading algorithms, models, and pipelines;
- Utilities: linear algebra, statistics, data handling, etc.

To learn more about programming with MLib please refer to the official documentation [6].

2.3.3. GraphX

GraphX[7] is a new component in Spark for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API[8].

To learn more about programming with GraphX please refer to the official documentation [7].

2.3.4. Spark Streaming

Spark Streaming[4] enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs. You can write Spark Streaming programs in Scala, Java or Python.

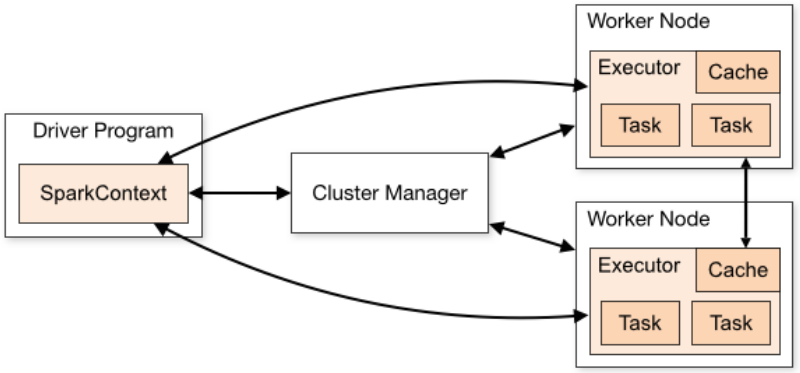
To learn more about programming with Spark Streaming please refer to the official documentation [4].

2.4. Running Spark on Clusters

As explained in the official documentation[9], Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program, called the driver program (See Figure 2, “Cluster Mode Overview (from https://spark.apache.org/docs/latest/cluster-overview.html) ”).

To run on a cluster, the SparkContext can connect to several types of cluster managers (e.g. Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run[9].

Figure 2. Cluster Mode Overview (from https://spark.apache.org/docs/latest/cluster-overview.html)



As explained in the official documentation[9], there are several useful things to note about this architecture:

- Each application gets its own executor processes. These stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (i.e. instances of SparkContext) without writing it to an external storage system.

- Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN). The driver program must listen for and accept incoming connections from its executors throughout its lifetime (e.g., see spark.driver.port in the network config section). As such, the driver program must be network addressable from the worker nodes.

- Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network.

The cluster managers Spark supports are listed in the official documentation at [9]. Some of these are:

- Standalone: a simple cluster manager included with Spark that makes it easy to set up a cluster.

- Apache Mesos: a general cluster manager that can also run Hadoop MapReduce and service applications.

- Hadoop YARN: the resource manager in Hadoop 2.

2.5. Spark Standalone Cluster

As explained in [10], you can launch a Spark standalone cluster by first creating a file called conf/slaves in your Spark directory. This file must contain the hostnames of all the machines where you intend to start Spark workers, one per line. If the file conf/slaves does not exist then only , a single machine (localhost) is used, which is useful for testing. Note, the master machine accesses each of the worker machines via ssh. By default, ssh is run in parallel and requires password-less (using a private key) access to be setup. If you do not have a password-less setup, you can set the environment variable SPARK_SSH_FOREGROUND and serially provide a password for each worker.

Once the conf/slaves file is set up you can launch or stop a cluster with the following scripts that are available in $SPARK_HOME/sbin [10].

- sbin/start-master.sh - Starts a master instance on the machine the script is executed on.

- sbin/start-slaves.sh - Starts a slave instance on each machine specified in the conf/slaves file.

- sbin/start-slave.sh - Starts a slave instance on the machine the script is executed on.

- sbin/start-all.sh - Starts both a master and a number of slaves as described above.

- sbin/stop-master.sh - Stops the master that was started via the sbin/start-master.sh script.

- sbin/stop-slaves.sh - Stops all slave instances on the machines specified in the conf/slaves file.

- sbin/stop-all.sh - Stops both the master and the slaves as described above.

Optionally it is possible to configure the cluster further by setting environment variables in conf/spark-env.sh. Create this file by starting with the conf/spark-env.sh.template, and copy it to all worker machines for the settings to take effect. For example,you can use it to select directories for logs and workers by setting the environment variables: SPARK_LOG_DIR and SPARK_WORKER_DIR.

2.6. Running Spark in Slurm

On a traditional HPC platform a Spark cluster can be run in standalone mode on top of a Slurm resource manager. This requires nodes exclusively allocated to run the Spark master and worker daemons. The `spark-submit` script can then be used to submit jobs to the Spark cluster.

Here is an example of the steps required to do this on one particular traditional HPC platform. Please note that you may need to customise the content of these steps for your own HPC platform.

1. Allocation of nodes in Slurm.

```
#SBATCH -o spark-pi.%j.%N.out
#SBATCH -e spark-pi.%j.%N.err
#SBATCH -D ./
#SBATCH -J spark-pi
#SBATCH --clusters=mpp2
#SBATCH --nodes=3
#SBATCH --mem 20000
#SBATCH --ntasks-per-node 4
#SBATCH --cpus-per-task 7
#SBATCH --time=00:10:00
```

2. Load software and start the master and workers. (Note the available modules on your HPC platform may differ and may need to install these yourself.)

```
source /etc/profile.d/modules.sh
module load java
module load python
module load R
module load spark
## Start master and slave in
spark-start
echo $MASTER
```

   The `spark-start` script starts the master and workers per Slurm task on the allocated nodes.

3. Launching a spark application in the spark cluster:

```
spark-submit --total-executor-cores 84 \
--executor-memory 5G \
$SPARK_HOME/examples/src/main/python/pi.py 1000
```

When submitting a Spark application there are a few tuning parameters which should be considered:

- the --ntasks-per-node parameter that specifies how many executors will be started on each node. By default, Spark will use 1 core per executor, thus it is essential to specify the --total-executor-cores, where this number cannot exceed the total number of cores available on the nodes allocated for the Spark application (84 cores resulting in 7 CPU cores per executor in this example).
- the --executor-memory parameter that specifies the memory per each executor. It is 2GB by default, and cannot be greater than a RAM available on a cluster node (64 GiB in allocated nodes for this example).

3. Cray Urika GX System

3.1. Introduction

The Urika GX is primarily targetted at users who wish to undertake highly interactive and iterative data analytics that require supercomputer levels of computing performance. Its architecture and supporting software stack therefore differs from a traditional high performance computing (HPC) platform.

In a traditional setting, a batch scheduler such as PBS or SLURM is employed to manage access to the computing resources available on an HPC platform. Typically, a user logs on to the front end node of the HPC platform and prepares a script that defines both the computing task they wish to execute and the amount of computing resources this task requires. The user then submits the script to the batch scheduler. The batch scheduler then determines when the script is executed. This approach allows many different users with different computing resource requirements to share an HPC platform. In addition, it allows efficient usage of the available computing resources.

However, this approach does mean that a user's job may have to wait for suitable resources to become available before it starts. Hence the user may have to wait some time, perhaps hours or even days, for their results. This makes it unsuitable for users who wish to have interactive access and who want to be able to immediately redirect their analyses based on up-to-date results. Moreover an HPC platform may not be configured appropriately for a particular user's needs instead it will be configured to match an overall optimum such as high throughput or capability. For example, a user may need a particular mix of compute and disk resources that few, if any other users, want. So the HPC service provider has little incentive to configure the HPC platform for such a user.

The Urika GX provides the users with the option to choose the type of resources (e.g. SSD) they wish to utilize for each of their applications, so that they will achieve the best possible performance. Moreover, Urika's resource manager can dynamically determine the optimal amount of resources that it should offer to every application so that the cluster's total resources are utilized optimally. In this way, not only can the platform address each individual user's needs, but it can also serve more users at the same time.

3.2. Overview

The Cray Urika GX system is an HPC platform dedicated to highly interactive and iterative data analytics that require supercomputer levels of computing performance. The chapter's contents are based on the Urika GX system hosted by EPCC and Cray on behalf of the Alan Turing Institute in the UK. This chapter contains sections on the following sections:

- the system's configuration (section 3.3);
- the way a user can access the system; (section 3.4);
- the production environment (section 3.5);
- the programming environment (section 3.6);
- the available performance analysis tools (section 3.7);
- the suggested parameter tuning (section 3.8);
- the debugging tools (section 3.9).

3.3. Architecture / Configuration

Section 3.3.1 describes the overall configuration of the Urika GX, while section 3.3.2 focuses on the node-level configurations. The specifications presented hold for all Urika-GX systems. The differences between the standard specifications and the EPCC-hosted platform will be explicitly noted.

3.3.1. System Level

At the time of writing Urika GX nodes use CentOS 7.2 operating system. These nodes are organized in GreenBlade chassis [11]. A Urika-GX system rack is depicted at figure 3.

Figure 3. Urika-GX system

This image is taken from https://www.cray.com/products/analytics/urika-gx

Urika GX has three kinds of networks:

Network	Description
Aries High Speed Network	This provides application and data connectivity between different nodes.
Operational Ethernet network	This is used for importing user data and accessing data streaming applications from compute nodes.
Management Ethernet network	This is used for system management.

3.3.2. Node Level

3.3.2.1. Types

There are three kinds of nodes in the Urika GX system:

Type	Usage
Compute Nodes	Applications and services are run on these nodes.
Login Nodes	A user logs in to these nodes and from there launches their applications onto the compute nodes.
I/O Nodes	These nodes handle the connections to external storage and file systems.

3.3.2.2. Configurations

3.3.2.2.1. Processors

Generally, Urika-GX systems use 2 processors per node, with the processors' type being one of the following:

- Intel Broadwell 18C E5-2697 v4
- Intel Broadwell 8C E5-2620 v4

As far as the EPCC-hosted Urika machine is concerned, it uses Intel Xeon E5-2695 v4. Each of the processors possess 18 cores, thus each node has 36 CPU cores.

3.3.2.2.2. RAM

Regarding the available memory per node, Urika systems offer three options:

- 128 GB
- 256 GB
- 512 GB

The EPCC-hosted machine has 256 GB per node.

3.3.2.2.3. Storage Memory

As far as the storage memory is concerned, all the nodes can have either 4 or 8 TB of HDD storage memory. On the other hand, SSD availability depends on the kind of the node:

Node type	SSD storage memory
Compute node	2 TB or 4 TB
Login node	Not available by default
I/O node	Not supported

For more information regarding the architecture and the configurations of Urika GX see [20].

3.4. System Access

3.4.1. Access to the EPCC-hosted Urika

Users of the EPCC-hosted Urika can obtain instructions on how to obtain a Urika account and access it by visiting http://ati-rescomp-service-docs.readthedocs.io/en/latest/cray/connecting.html .

3.4.2. User Interfaces

3.4.2.1. Access to User Interfaces

Most of the user interfaces can be accessed through the primary Urika-GX Applications Interface (UAI). An alternative way is to use the following urls:

User Interface	**URL**
Urika-GX Applications Interface	`http://urika1.turing.ac.uk:80`
YARN Resource Manager	`http://urika1.turing.ac.uk:8088`, `http://urika2.turing.ac.uk:8088`
Hadoop Job History Server	`http://urika1.turing.ac.uk:19888`, `http://urika2.turing.ac.uk:19888`
Marathon	`http://urika1.turing.ac.uk:8080`, `http://urika2.turing.ac.uk:8080`
Mesos Master	`http://urika1.turing.ac.uk:5050`, `http://urika2.turing.ac.uk:5050`
Spark Application's Web UI	`http://urika1.turing.ac.uk:4040`, `http://urika2.turing.ac.uk:4040`. In case multiple applications are launched, they run on ports `4041`,`4042`,`4043` onwards.
Spark History Server	`http://urika2.turing.ac.uk:18080`, `http://urika2.turing.ac.uk:18080`
Grafana	`http://urika2.turing.ac.uk:3000`
Jupyter Notebook	`http://urika1.turing.ac.uk:7800`
Cray Application Management	`http://urika1.turing.ac.uk/applications`

Whenever a user accesses an application user interface (e.g. Spark, Grafana) though the UAI, a banner containing learning resources and other links is also visible. Here users can find the Urika system documentation and guides through the learning resources link. Moreover, users are provided with tutorials on the software pre-installed on the Urika GX. This banner is not visible when the users choose to access the user interfaces using the URLs listed in the table above.

The *Urika-GX Analytic Applications Guide* [19] contains further URLs for the user interfaceses of other Urika GX applications.

More information regarding the role of each user interface on the Urika GX is given in section 3.5.

3.4.2.2. Authentication

The following authentication mechanisms can be used to access certain user interfaces:

User Interface	**Username**	**Password**
Grafana	admin	`admin`
Jupyter Notebook	login_username	`login_password`

User Interface	Username	Password
Mesos Master	login_username	The password can be found in `/security/secrets/userName.mesos` file.
Jupyter Notebook	login_username	`login_password`
Cray Application Management UI	LDAP	`username : admin , password: admin`

The *Urika-GX Analytic Applications Guide* [19] contains the authentication mechanisms for further Urika GX user interfaces.

3.4.3. Data Transfer (EPCC-hosted Urika)

Users of the EPCC-hosted Urika can use secure copy (i.e. scp) to transfer to and from their Urika GX. Instructions on how to scp to do this can be found at http://ati-rescomp-service-docs.readthedocs.io/en/latest/cray/connecting.html.

The SFTP network protocol is also supported (see `$ man sftp`).

3.5. Production Environment

Sections 3.5.1, 3.5.2 and 3.5.3 describe the components of Hadoop, Spark and Cray Graph Engine respectively, provided on the Urika GX platform. The way Urika manages resources is presented in section 3.5.4, while the different types of file systems are presented in section 3.5.5. Finally, the way fault tolerance is preserved and user interfaces can be utilized is presented in sections 3.5.6 and 3.5.7 respectively.

3.5.1. Hadoop and its ecosystem

The Urika GX ships with Hortonworks Data Platform (HDP) which includes Apache Hadoop. Apart from the core Hadoop components, the following Hadoop ecosystem components are installed on Urika GX:

Component	**Description**
Apache Avro	The data serialization system: as explained at [12] when Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program.
Apache DataFu	This is a collection of libraries for working with big data on Hadoop. For more information see https://datafu.apache.org/.
Hive	As explained at [13] this is a data warehouse system that uses SQL to read, write and manage large datasets residing in distributed storage. Structure can be projected onto data already in storage.
Hue	This is a visual interface or workbench for querying and visualizing data.
Apache Kafka	As explained at [14] this is distributed streaming platform that enables publishing of and subscribing to streams of records. It stored these records in a fault-tolerant way and enables processing of these records as they occur.
Apache Oozie	This is a workflow scheduler system for managinh Hadoop jobs. For more information see http://oozie.apache.org/.
Apache Parquet	As explained at [15] this is columnar format data storage on Hadoop.
Apache Pig	As explained at [16] this a platform for analyzing large data sets. It consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The key feature of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets.
Apache Sqoop	This is a tool for transferring bulk data between Hadoop and structured data stores. For more information see http://sqoop.apache.org/.
Apache HiveServer2	This service enables clients to executing queries against against Hive.
Apache Hive Thrift Server	RPC framework for building cross-platform services
Apache Zookeeper	Configuration manager for distributed systems

More information on these components can also be found at http://hortonworks.com and http://www.apache.org.

3.5.2. Apache Spark

Urika GX supports the following Spark core and ecosystem components:

- Spark Core, DataFrames, and Resilient Distributed Datasets (RDDs)
- Spark SQL, Datasets and Dataframes
- Spark Streaming
- MLlib Machine Learning Library
- Spark Streaming
- GraphX

More information with regards to the above components can be found at https://spark.apache.org/documentation.html and section 2.

3.5.3. Cray Graph Engine

Cray Graph Engine (CGE) is a software application capable of searching large graph-oriented databases and querying complex relationships between data items. CGE is designed to store and analyze datasets when the patterns of relationships and interconnections between data items are at least as important as the data items themselves. It includes two major components:

Component	**Description**
Graph Oriented Database	Database that uses graph structures to store and represent data.
Resource Description Framework (RDF)	Data representation standard, presenting data as a triple containing a subject, a predicate and an object.

As opposed to the storage technique of the relational databases, CGE uses RDFs to store data. For more information regarding RDFs read section *2.2.2* from *Cray Graph Engine User Guide* [18].

3.5.4. Resource Management

Urika GX possesses a number of different resource management tools. Urika's resource management enables system resources to be allocated dynamically, based on the needs of each application.

3.5.4.1. Mesos

Mesos acts as the primary resource manager on Urika-GX and lies between the operating system and the application layer. Its task is to optimize resource utilization.

Mesos does not decide about the schedule and execution of the different jobs. Moreover, Mesos does not offer a queue. Instead, Mesos offers resources to the frameworks that are registered with it. It is up to the framework's scheduler to decide whether to accept or reject the offer. If the offer is accepted, it is the framework's responsibility to schedule the execution of the jobs, using the resources provided. In case when the offer is rejected, Mesos will continue to make new offers based on resource availability. On the Urika the frameworks available with Mesos, are as follows:

- Marathon (for more information section 3.5.4.2)
- Yarn (for more information see section 3.5.4.4)
- Each Spark job (for more information see section 3.5.4.5)

The Mesos architecture consists of the following components:

- Mesos agents/slaves
- Mesos masters

Mesos slaves play the role of the cluster's resources. Mesos master decides how many resources to offer to each framework, according to an organizational policy (e.g. fair sharing or priority). The reasons why Mesos ships with more than one master are presented in section 3.5.6. Mesos masters are configured with Apache Zookeeper.

3.5.4.2. Marathon

Marathon is used for launching long-running applications to run under Mesos and acts as a Mesos ecosystem component. Marathon is registered as a single framework with Mesos. Marathon's API is not capable of determining if there are enough resources for a job that has not been submitted. Therefore, Marathon uses Mesos to negotiate for resources. When Mesos informs Marathon that the required resources are available, the job is posted to Marathon. Marathon instances are also configured with Zookeeper.

3.5.4.3. Mrun

Mrun is a Cray-developed application launcher, which is built upon Marathon commands. It uses Marathon in order to set up resources for CGE and HPC jobs. Mrun is submitted as an application to Marathon, therefore no job is posted until the resource requirements are satisfied. Mrun needs to be executed by a login node. It cannot be executed by a tenant VM. Finally, if an Anaconda environment is activated on the login node when mrun is used, compute nodes are aware of that virtual environment.

The following commands are used to obtain information about the status of Mesos and Marathon environment:

Command	**Description**
`$ mrun --info`	This is used to obtain a snapshot of the active frameworks registered with Mesos, Marathon applications and the available computing resources.
`$ mrun --resources`	This provides with a list of the system's nodes along with their availability status and their CPU/memory specifications.

The following commands are used to launch HPC applications:

Command	**Description**
`$ mrun app.exe`	*app.exe* will run as a single task on one node.
`$ mrun -n num_of_tasks \ -N num_of_nodes app.exe`	*app.exe* will run as *num_of_tasks* tasks on *num_of_nodes* nodes.
`$ mrun -n num_of_tasks \ -N num_of_nodes app.exe \ --wait \ --immediate=num_of_seconds`	*app.exe* will run as *num_of_tasks* tasks on *num_of_nodes* nodes. If the required resources are not available instantly, mrun will continue to poll Mesos. Providing that the required resources become available within *num_of_seconds* seconds, the application will be posted to Marathon. Otherwise, mrun will time out.

The following commands are used with regards to a specific running Marathon application:

Command	**Description**
`$ mrun --detail appID`	This outputs additional information with regards to the application with ID: *appID*
`$ mrun --cancel appID`	This cancels or aborts the application with ID: *appID*

For more information, read the manual of mrun (`$ man mrun`)

3.5.4.4. Yarn

Yarn acts as the resource manager for Hadoop jobs on Urika GX and uses its own queue for Hadoop workloads. Cray has developed scripts to set up resources for Yarn. These scripts are submitted as applications to Marathon and they allow the dynamic allocation of resources between Mesos and Yarn. Just like mrun, Yarn scripts cannot be executed if the required resources are not available. When the requested nodes are not available, the current resource availability is reported and the script exits. Yarn scripts are also known as *flex scripts* and are presented below:

Command	Description
`$ urika-yam-status`	This displays the lists of existing applications and the resources allocated to each application. For more information, read the manual of urika-yam-status (`$ man urika-yam-status`).
`$ urika-yam-flexup \ --nodes num_of_nodes \ --identifier request_id \ --timeout num_of_minutes`	This is used to 'flex up' *num_of_nodes* nodes. *request_id* is the unique identifier of the request. In case the request is accepted and the flexed up nodes are idle for *num_of_minutes* the resources will be automatically released. For more information read the manual of urika-yam-flexup (`$ man urika-yam-flexup`).
`$ urika-yam-flexdown \ --identifier request_id`	This is used to manually 'flex down' the nodes flexed up by the request with ID: *request_id* . For more information, read the manual of the urika-yam-flexdown command (`$ man urika-yam-flexdown`).

The following command is used for launching a hadoop application:

Command	Description
`$ yarn jar file.jar \ main_class \ arg1 arg2 arg3`	This is used to run job: *file.jar*, while *main_class* is the main class of the executable and *arg1 arg2* and *arg3* are command line arguments of the program. The main class and the command line arguments are optional parameters.

Now we will present an example of a Hadoop job submission. In this example we will run a TeraSort benchmark. Given the fact that we have already accessed a login node, we will have to use the following commands:

1. Checking whether there are available nodes to flex up:

   ```
   $ mrun --resources
   ```

2. Checking whether we have already flexed up some nodes and how many nodes are flexed up for the needs of other applications:

   ```
   $ urika-yam-status
   ```

3. Flexing up 3 nodes for the needs of our job:

   ```
   $ urika-yam-flexup --nodes 3 --identifier hadoopexample
   ```

4. If the folder expected to store the output of our Hadoop jobs already exists, it has to be deleted before the job's submission:

   ```
   $ hdfs dfs -rm -R /tmp/10gsort
   ```

5. This job generates the data which will be used as an input for TeraSort:

   ```
   $ yarn jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-mapreduce-examples.jar teragen 100 /tmp/10gsort/input
   ```

6. Executing TeraSort benchmark:

```
$ yarn jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-mapre-
duce-examples.jar terasort /tmp/10gsort/input /tmp/10gsort/output
```

7. This job evaluates the output of the TeraSort benchmark:

```
$ yarn jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-mapre-
duce-examples.jar teravalidate /tmp/10gsort/output /tmp/10gsort/validate
```

8. Confirming the success of the validation by checking the output of the validation job:

```
$ hdfs dfs -ls /tmp/10gsort/validate
```

9. Flexing down the nodes that we used:

```
$ urika-yam-flexdown --identifier hadoopexample
```

3.5.4.5. Spark Jobs

Spark, like Marathon, is preconfigured to authenticate with Mesos. Each spark job is registered as a separate framework with Mesos. However, Spark jobs do not behave in the same way Mrun and Yarn scripts do. Spark can accept offers with fewer resources than it is requested.

In order to connect to Mesos, Spark master is set to:

```
mesos://zk://zoo1:2181,zoo2:2181,zoo3:2181/mesos
```

Spark launch wrapper scripts are used for the launch of spark applications or interactive shells. These scripts are located to:

```
/opt/cray/spark2/default/usrScripts
```

The provided spark launch wrapper scripts are the following:

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

The user can use the following flags to change the default settings of the above scripts:

Flag	**Description**
`--total-executor-cores`	This sets the number of desired cores.
`--driver-memory`	This sets the desired amount of memory allocated to the driver. By default 16 gigabytes are allocated to the driver.
`--executor-memory`	This sets the desired amount of memory allocated to the executors. By default 96 gigabytes are allocated to each executor.

The users also have the option to use both SSDs and HDDs (instead of SSD alone) in order to provide their spark jobs with additional temporary space. In order to achieve that, they will have to follow the steps below:

1. Create a file named `spark_local_dirs.hdd` under their home directory (`/home/users/`*`user-name`*`/spark_local_dirs.hdd`)

2. Use the command `echo true >> /home/users/`*`username`*`/spark_local_dirs.hdd` to add true to the file's contents.

In case the users wish to revert to the default configurations they just have to delete `spark_local_dirs.hdd` .

For more information on resource management on Urika read *Urika-GX Analytic Applications Guide* [19].

3.5.5. File Systems

3.5.5.1. Hadoop Distributed File System (HDFS)

HDFS is a highly fault tolerant distributed file system. Hadoop uses HDFS to store data. The Urika GX also has tiered HDFS data storage. HDFS data is stored on the SSDs and HDDs of Urika GX's compute nodes and is transfered over the Aries Network. HDFS is the data store for all the Hadoop components on Urika GX. Users cannot have write access to HDFS unless an administrator has provided them with a designated folder under `hdfs:///user` .

3.5.5.2. Network File System (NFS)

NFS is a distributed file system protocol. NFS is made available to every node via the management network. NFS is not suitable for big data transfers and large writes, as this will cause the network to operate much slower and timeout. Home directories are mounted on NFS, with limited space.

3.5.5.3. Lustre

Lustre is a parallel distributed file system. It is suitable for larger data sets and it is supported as an external file system on Urika GX. Lustre is mounted at `/mnt/lustre`.

For more information on Urika's file systems read *Urika-GX Analytic Applications Guide* [19].

3.5.6. Fault Tolerance

Urika GX is fault tolerant and so provides resiliency against system failures. Failed jobs are re-scheduled automatically.

3.5.6.1. Zookeeper

Zookeeper enables highly reliable distributed coordination. On Urika, 3 Zookeeper instances are running, while a minimum of 2 are always available. Urika uses Zookeper to provide Mesos and Marathon with fault tolerance.

3.5.6.2. Hadoop

Whenever there is a failure in the execution of a Hadoop job, the corresponding process is reported to the master and is re-scheduled.

3.5.6.3. Spark

Spark tracks transformations and actions through an acyclic lineage graph. In case of a failure, Spark detects the point of failure and re-schedules the after-the-failure computations to a different node.

3.5.6.4. Mesos

Mesos runs on high availability mode. Similarly to Zookeper, Mesos has 3 master instances running. If one of them fails, one of the remaining two is elected as the new master. In this way, no disturbance takes place during the resource management process.

3.5.6.5. Marathon

Marathon also has 3 running instances and follows the same procedure with Mesos in case one of these instances fails. If a Mesos task fails, Marathon will accept more resources from Mesos and another task will be launched, usually on a different node.

For more information regarding Urika's fault tolerance read *Urika-GX Analytic Applications Guide* [19].

3.5.7. User Interfaces

Section 3.4 contains information regarding the different ways to access the most important user interfaces featured on Urika GX.

3.5.7.1. Urika GX Applications Interface (UAI)

UAI is the primary entry point to view a number of applications running on Urika GX. Moreover, it is used for accessing training material and monitor the system's health information.

3.5.7.2. Grafana

Grafana is a metrics, dashboard, and graph editor. Grafana can be used for the monitoring of system resources.

Two of the major components of Grafana are the following:

- *Organizations* correspond to different deployment models.
- *Users* are named accounts in Grafana.

A user can belong to one or more organizations. Furthermore, a user can have different privileges, depending on the role he has been assigned to.

For information regarding the performance analysis tools of Grafana, visit section 3.7.

3.5.7.3. Cray Application Management UI

The Cray Application Management UI contains information about both running and finished jobs. This UI enables users to access the logs of Spark jobs or delete jobs they have submitted.

3.5.7.4. Spark

Each Spark application launches its own Web UI. This UI can be used to monitor running Spark jobs and displays useful information, such as scheduler stages/tasks, RDD sizes/memory usage etc. On the other hand, the Spark History Server monitors completed Spark jobs.

Both of these UIs link Spark applications to the Grafana UI, where more information regarding resource utilization is displayed.

3.5.7.5. Hadoop

Hadoop jobs can be monitored using the following three interfaces:

- Hadoop Job History Server
- YARN Resource Manager
- Cray Application Management UI

3.5.7.6. Mesos

Mesos Web UI can be used to monitor different components of the Mesos cluster, such as the Mesos slaves, resources and frameworks. Users can use Mesos Web UI to view the resources reserved as well as their tasks. Users should avoid launching applications directly from Mesos Web UI.

3.5.7.7. Marathon

Marathon Web UI can be used for the creation of applications. Users should avoid deleting the analytic applications that use the 'flex scripts', except if it is mandatory to shut down nodes used by Yarn.

For more information regarding the available Urika UIs read *Urika-GX Analytic Applications Guide* [19].

3.6. Programming Environment

Section 3.6.1 explains the basic programming components of Urika GX, while section 3.6.2 describes how Anaconda can be used. In section 3.6.3 Jupyter Notebook is presented, while in the last section (3.6.4) the programming options offered by Spark are explained.

3.6.1. Components

Some of the components of Urika's analytics environment are presented below:

- Python 2
- Python 3
- Scala
- R
- Anaconda Python
- Numpy
- Scipy
- Git
- gcc
- Apache Maven

In addition to the above components, Urika also has a number of enviromental modules.

3.6.2. Anaconda

Anaconda contains conda, which is a source package and environment manager. Anaconda enables users to easily install pre-compiled software locally, without needing administrator privileges. A user can use the following commands in order to load and perform basic management of anaconda's environments:

Command	**Description**
`$ module load anaconda3`	The `anaconda3` module is loaded and Anaconda Python becomes the default Python.
`$ conda create --name py36Env \ python=3.6`	A new environment with Python 3.6 is created.
`$ source activate py36Env`	The conda environment `py36Env` is activated. Both PySpark and Python utilize the active environment.
`$ source deactivate`	The active conda environment is deactivated.
`$ module unload anaconda3`	The `anaconda3` module is unloaded.

For more information regarding the management of Anaconda environments, the users can visit https://docs.anaconda.com/anaconda/navigator/tutorials/manage-environments.

3.6.3. Jupyter Notebook

Jupyter Notebook is a a web application that creates executable documents. Moreover, it enables adding explanatory text between executable cells.

On Urika GX, Jupyter Notebook supports by default the following kernels:

- Python 2
- Python 3
- Bash
- R
- PySpark
- Scala
- SparkR

Jupyter Notebook's users might usually need to use python libraries and packages that are not provided by the default python kernels. In this case, they are able to create a new customizable ipython kernel though an Anaconda environment. We present an example below:

Command	**Description**
`$ module load anaconda3`	The `anaconda3` module is loaded.
`$ conda create --name jupyterEnv \ python=3.6`	A new environment with Python 3.6 is created.
`$ conda install \ --name jupyterEnv ipykernel`	`ipykernel` is installed under `jupyterEnv` environment.
`$ source activate jupyterEnv`	`jupyterEnv` environment is activated.
`$ python -m ipykernel install \ --user --name jupyterEnv \ --display-name "My Python Kernel"`	Python kernel `My Python Kernel` is created by the `jupyterEnv` environment.

After the execution of the commands above `My Python Kernel` is added to the kernel options provided by the Jupyter Notebook User Interface. This kernel is able to utilize every python package installed under `jupyterEnv` Anaconda environment. Moreover, the user does NOT have to activate `jupyterEnv` everytime `My Python Kernel` is to be used.

A user must stop their notebooks before they log off or else the notebook will continue to use resources unnecessarily. In cases where Jupyter processes are still running after a user has logged out, the Linux `kill` command can be used to manually kill them.

3.6.4. Apache Spark

The `spark/2.3.0` environmental module is loaded by default after a user logins to a login node. On Urika GX, Spark comes with APIs for Java, Scala, Python and R.

3.6.4.1. Java

Java applications are built using Maven. The following dependency should be added to the `pox.xml` file, similarly to the following example:

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

```
</dependencies>
```

3.6.4.2. Scala

Scala applications are built using Scala Build Tool (sbt). A dependency, like the one presented below, should be added to .sbt file.

```
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
```

A user can set the required number of Spark cores using Scala code. If `NUM_CORES` is the required number of Spark cores, the next lines should be added to Spark's command line (after invoking `spark-shell` wrapper script) or to Jupyter Notebook.

```
sc.stop()
sc = SparkContext(conf=SparkConf().set("spark.cores.max", "NUM_CORES"))
```

3.6.4.3. PySpark

PySpark is aware of Anaconda environments. In case there is one activated anaconda environment, Spark will utilize the version the environment uses. In order for the anaconda's Python version to be overridden, `PYSPARK_PYTHON` environmental variable should be manually set to point to the required Python version.

It is possible for the user to change the default number of Spark cores required, by adding the following lines to Spark's command line (after invoking the `pyspark` wrapper script) or to Jupyter notebook:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
val conf = new SparkConf().set("spark.cores.max", "NUM_CORES.")
val sc = new SparkContext(conf)
```

`NUM_CORES` is the required number of Spark cores.

3.6.4.4. SparkR

SparkR can also be used to set the required number of Spark cores used for a Spark application. The following commands should be added to Spark's command line (after `SparkR` wrapper script has been invoked) or to Jupyter notebook:

```
sparkR.session(spark.cores.max = "NUM_CORES")
```

3.7. Performance Analysis

Urika's main performance analysis tool is Grafana. Grafana dashboards collect all the visualizations into an individual interface and they include:

- Statistical data regarding network, I/O and CPU utilization both for every node and the system as a whole.
- Metrics regarding Hadoop applications and cluster.
- Statistical data regarding Spark jobs.

3.8. Tuning

Spark's configurations on Urika GX have some differences from Spark's standard configurations:

- `spark.shuffle.compress = false` and `spark.locality.wait = 1` . These configurations result in a better performance for some applications on Urika GX. In case an application is running out of memory or SSD space, `spark.shuffle.compress` should be switched back to `true` .
- Each executor is provided with 96GB of memory, while the driver is provided with 16GB.

By default, Spark runs temporary files on the SSDs of the compute nodes. However, a combination of HDDs and SSDs offers flexibility, especially when a Spark job requires large shuffle space. On the other hand, using only SSDs provides the best performance. For information on how to change the default storage of Spark's temporary files read section 3.5.4.5.

In section 3.5.4 we referred to the fact that Spark jobs accept offers with less resources than the ones requested. Users can control the minimum of resources that a spark application can accept through the variable `spark.scheduler.minRegisteredResourcesRatio` .

Section 20.4 of *Urika-GX Analytic Applications Guide* [19] refers to more tunable Spark and Hadoop configuration parameters.

3.9. Debugging

In section 3.9.1 we refer to the components of Urika that can be used as debugging tools, while in section 3.9.2 we present the location of the log files of different applications of Urika GX.

3.9.1. Tools

In section 3.5.7 we presented Hadoop Job History Server UI and Yarn Resource Manager UI, which can be used for the monitoring of Hadoop applications. Regarding Spark jobs, Spark Web UI and Spark History Server can help during the debugging process, while the Spark shell can also be an effective debugging tool. Cray Application Management UI can also be used for the monitoring and the debugging of Hadoop, Spark and CGE jobs.

3.9.2. Log Files

Apart from presenting the status of applications, Cray Application Management UI provides with links to the generated log files of the various jobs. The physical location of log files of some applications are given below:

Application	**Log File Location**
Mesos	`/var/log/mesos`
Marathon	`/var/log/messages`
Grafana	`/var/log/grafana/grafana.log`
Hadoop	`/var/log/hadoop/hdfs/` and `/var/log/hadoop/yarn/` of the individual compute nodes. `/app-logs` in hdfs.
Spark	`/var/log/mesos/agent/slaves/` of the individual compute nodes.
Jupyter Notebook	`/var/log/jupyterhub.log`
Flex Scripts	`/var/log/urika-yam.log`

For more information regarding debugging, log files and troubleshooting of the various Urika applications read *Urika-GX Graph Engine User Guide* [18] and *Urika-GX Analytic Applications Guide* [19].

Further documentation

Books

[1] *Best Practice Guide - Intel Xeon Phi, January 2017, http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi-1.pdf .*

Websites, forums, webinars

[2] *PRACE Webpage, http://www.prace-ri.eu/.*

[3] *Apache Spark, https://spark.apache.org/docs/latest/index.html.*

[4] *Apache Spark Streaming, https://spark.apache.org/docs/latest/streaming-programming-guide.html.*

[5] *Apache Spark SQL, https://spark.apache.org/docs/latest/sql-programming-guide.html.*

[6] *Apache Spark MLib, https://spark.apache.org/docs/latest/ml-guide.html.*

[7] *Apache Spark GraphX, https://spark.apache.org/docs/latest/graphx-programming-guide.html.*

[8] *Pregel: a system for large-scale graph processing, Proceeding SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data Pages 135-146, Indianapolis, Indiana, USA — June 06 - 10, 2010 .*

[9] *Cluster Mode Overview, https://spark.apache.org/docs/latest/cluster-overview.html.*

[10] *Spark Standalone Mode, https://spark.apache.org/docs/latest/spark-standalone.html.*

[11] *Urika-GX Hardware Guide (Rev C.) H-6142, https://pubs.cray.com/content/00485604-DB/FA00242577.*

[12] *Apache Avro 1.8.2 Documentation, http://avro.apache.org/docs/current/.*

[13] *Apache Hive, https://hive.apache.org/.*

[14] *Apache Kafka, https://kafka.apache.org/intro.*

[15] *Apache Parquet, https://parquet.apache.org/.*

[16] *Apache Pig, https://pig.apache.org/.*

Manuals, papers

[17] *PRACE Public Deliverable 7.6 Best Practice Guides for New and Emerging Architectures, http://www.prace-ri.eu/IMG/pdf/D7.6_4ip.pdf.*

[18] *Cray® Graph Engine User Guide (3.1.UP02) S-3014 .*

[19] *Urika®-GX Analytic Applications Guide (2.0.UP00) S-3015 .*

[20] *Urika®-GX System Overview (2.0.UP00) S-3017.*