Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Integration and Performances of Spark on a PBS-based HPC Environment

Francesco Gargiulo, Stefano Silvestri, Gennaro Oliva, Mario Ciampi

RT-ICAR-NA-2018-01 **Maggio 2018**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Napoli, Via P. Castellino 111, I-80131 Napoli, Tel: +39-0816139508, Fax: +39-0816139531, e-mail: napoli@icar.cnr.it, URL: *www.na.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Integration and Performances of Spark on a PBS-based HPC Environment

Francesco Gargiulo, Stefano Silvestri, Gennaro Oliva, Mario Ciampi

Rapporto Tecnico N: **RT-ICAR-NA-2018-01** ***Data:*** **Maggio 2018**

Integration and Performances of Spark on a PBS-based HPC Environment

Francesco Gargiulo, Stefano Silvestri, Gennaro Oliva, Mario Ciampi
Institute for High Performance Computing and Networking of the National Research Council of Italy (ICAR-CNR)
Via Pietro Castellino, 111 – 80131 Naples, Italy
{francesco.gargiulo, stefano.silvestri, gennaro.oliva, mario.ciampi}@icar.cnr.it

Abstract

In this technical report we analyse a methodology to integrate the Apache Spark Framework into a TORQUE HPC batch environment. We will show the pro and cons to integrate SPARK within TORQUE, underlining the compatibility issues and showing how to overcome the most common problems. We will also give the main technical details about the use of the aforementioned architecture. In the experimental assessment we also take into account the BLAS library advantages and we describe how to install/configure them properly to correctly run within the architecture. Finally, we will show the empirical results using a very challenge use case, namely the realization of a document search engine using the whole PubMed literature, aiming at the evaluation of the performances of this kind of architecture.

Keywords: Spark, HPC, TORQUE, Blas, Mesos, Yarn, Big Data Analytics.

1. Introduction

One big challenge faced by our society is a deluge of Big Data, which is an important field in High Performance Computing (HPC). Apache Spark[1] [1] is an open-source cluster-computing framework able to solve many issues in Big Data processing. Actually Spark is very popular among the scientific community, due to its high performances; in fact, it was developed in response to limitations in the MapReduce cluster computing paradigm, providing up to 100 times better performance for some specific applications [2].

Spark is implemented on dedicated commodity clusters, but the data storage and computational requirements are expected to increase as the workload and dataset sizes increase. Due to economies of scale, it will be far more efficient if Spark users can be given a usable environment on mainstream parallel computing resources at existing HPC centers, where both machines and people are already in place to assure that top-notch resources and support are readily available. In this technical report we illustrate the approach used to integrate Spark into HPC environments using the popular TORQUE [3] batch management system.

The integration of cluster-based analytics software such as Spark with an HPC batch environment like TORQUE has a number of potential benefits. First, it allows the analytics software to coexist with other applications and leverage the investments made for more traditional HPC applications, rather than requiring a completely separate analytics platform with the additional attendant costs for hardware, maintenance, power, cooling, and management staff. Second, running cluster-based analytics software in an HPC batch environment allows for relatively location management, which neither YARN nor Mesos currently provides.

[1] https://spark.apache.org/

Third, it allows for automated workflows that encompass both analytics and traditional HPC applications such as simulation and visualization. Most HPC sites prefer that parallel programs be launched using facilities provided by the batch environment.

The installation and management of Spark cluster environment on a classical HPC cluster system can led to obtain all aforementioned advantages, but there are not trivial operations to be performed to make the cluster run in the correct way. In [4], it is described how to integrate Spark into Portable Batch System (PBS)-based HPC environments; extending this work, in this technical report we describe how to exploit all advantages of this hybrid architecture, highlighting the pro and cons, underlining all the compatibilities issues and evaluating the performances on a Big-Data test set. The obtained results have been even successfully used in the development of the experiments described in [5] and [6].

For testing and assessing this hybrid environment we use the scientific biomedical literature database from PubMed[2]. PubMed is a free resource developed and maintained by the National Center for Biotechnology Information (NCBI), a division of the U.S. National Library of Medicine (NLM), at the National Institutes of Health (NIH). It actually comprises over 27 million citations and abstracts for biomedical literature indexed in NLM's MEDLINE database, as well as from other life science journals and online books. PubMed citations and abstracts include the fields of biomedicine and health, and cover portions of the life sciences, behavioral sciences, chemical sciences, and bioengineering. PubMed also provides access to additional relevant websites and links to other NCBI resources, including its various molecular biology databases. PubMed uses NCBI's Entrez search and retrieval system. PubMed does not include the full text of the journal article; however, the abstract display of PubMed citations may provide links to the full text from other sources, such as directly from a publisher's website or PubMed Central (PMC).

This technical report is organized as follows. In the Section 2 a general description of Apache Spark will be given, focusing on the Data Structure used and the main components involved; in Section 3 a detailed description of the Spark entities involved into the Spark cluster architecture will be described; Section 4 will be devoted to an overview of the possible cluster manager; in Section 5 a PBS-Based HPC environment description with the details on how to configure and use Spark over a Torque cluster system will be given. In Section 7 we will show the experimental results obtained using a Big Data source such as PubMed data and, finally, in Section 8 the conclusion will be pointed out.

2. Apache Spark

Apache Spark is a cluster computing platform designed to be a fast and general cluster computing system for analysing large amounts of data on clusters such as Big Data. Starting from Hadoop MapReduce, it extends its model to efficiently use it for more types of computations, which include interactive queries and stream processing. Spark offers a lot of benefits, as it can perform operations directly in Central memory and it can deliver performance even 100 times higher than MapReduce on specific applications [2].

It is easy to be used by software programmers, it is fast in the Big Data analyses and it also offers extensibility and interactive analytics. The Spark project contains multiple closely integrated components, supporting text files, Sequence Files, Avro, Parquet, and any other Hadoop InputFormat.

Spark is written in Scala Programming Language and it is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and a rich set of built-in libraries [7]. In our work Spark runs on an HPC cluster, where access is regulated by the Torque [3] resource manager. Working data sets are loaded from the file system, cached and computed upon repeatedly. It is well suited for large data analysis, caching data sets in memory.

There are many reasons to choose Spark to analyze the Big Data:

[2] https://www.ncbi.nlm.nih.gov/pubmed/

- unified engine for diverse workloads;
- easy to get started;
- leverages the best performances in distributed batch data processing;
- rich and complete Standard Library;
- works expressed in RDD - Distributed Parallel Scala Collections;
- unified development and deployment environment;
- ETL (Extract, Transform and Load) done easier;
- fault tolerance management.

2.1 Data structure of Spark: Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. An RDD is a distributed collection of elements that allows for the loading and manipulation of Big Data datasets in memory among many nodes. In Spark all works are expressed with new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. Spark uses the concept of RDD to achieve faster and efficient MapReduce operations. Spark's RDDs are recomputed each time an action on them is run. Users create RDDs in two ways: loading an external dataset, or distributing a collection of objects in their driver program. In summary, the RDD is a read-only, partitioned collection of records; RDD is a fault-tolerant collection of elements that can be operated in parallel execution [8].

RDDs offer two types of operations: transformations and actions:

- transformations construct a new RDD from a previous one;
- actions compute a result based on an RDD, and either return it to the driver program or save it to an external storage system.

2.2 Spark Architecture and Component Set

In the figure below the main components of Spark architecture are represented. The first layer consists on the cluster managers (see section 4). The middle layer, *Spark Core*, manages and controls the data and the execution flow of the components on the upper layer of the architecture. In the following of this section some details about the second and third layers of the architecture are provided, while the cluster managers are discussed in details in Section 4.

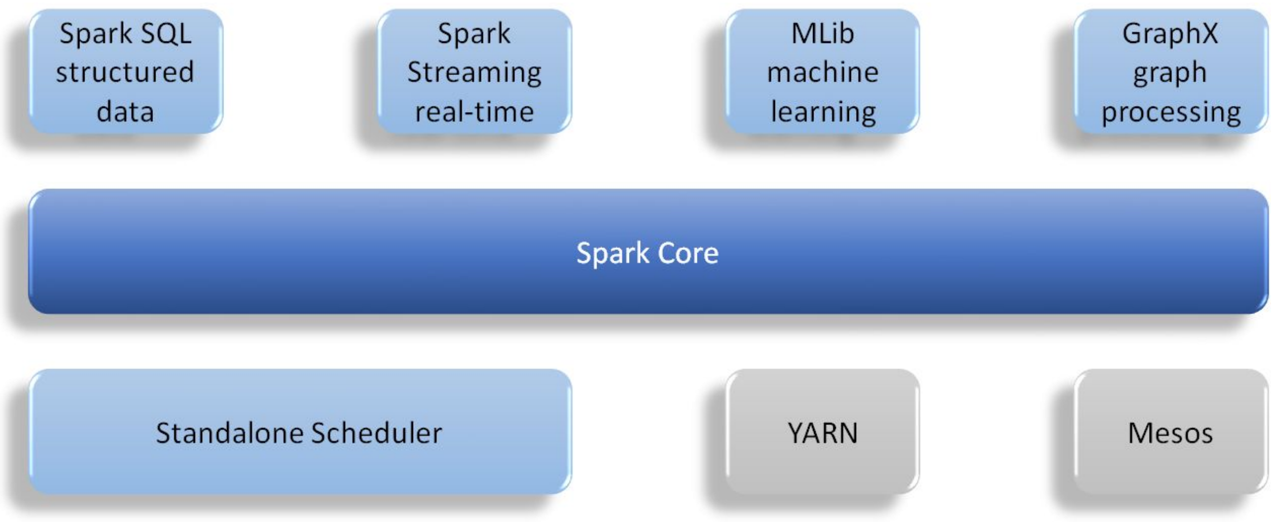


Figure 1 - Spark architecture components

2.2.1 Spark Core

Spark Core is a distributed execution engine, which contains the basic functionality of Spark:

- components for task scheduling;
- memory management;
- fault recovery;
- interaction with storage systems.

Spark Core contains also the API that defines RDDs. In particular, Spark Core provides many APIs for building and manipulating the RDD distributed across many compute nodes that can be manipulated in parallel.

2.2.2 Spark SQL

Spark SQL is a Spark package for working with structured data; it allows the users to realize ETL of their data from different formats, transform it, and expose it for ad-hoc querying.

Spark SQL provides three main capabilities:

- loading data from a variety of structured sources;
- querying the data using SQL, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors;
- when used within a Spark program, Spark SQL provides rich integration between SQL and regular Python/Java/Scala code.

The formats of the data are:

- **Parquet**, a popular column-oriented storage format that can store records with nested fields efficiently. It is often used with tools in the Hadoop ecosystem, and it supports all of the data types in Spark SQL. Spark SQL provides methods for reading data directly to and from Parquet files.
- **JSON** format can be managed by Spark SQL inferring automatically the schema by scanning the file, giving the possibility to access the fields of the records by name.

2.2.3 Spark Streaming

Spark Streaming is a Spark component that enables processing of live streams of data. Spark Streaming receives data streams and divides the incoming data into batches using a time interval defined by the user. It provides an API for manipulating data streams that closely match the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real-time [9].

Spark is built on the concept of RDD and Spark Streaming provides an abstraction called DStreams, or discretized streams. DStreams can be created from various input sources; they offer two types of operations: transformations, which yield a new DStream, and output operations, which write data to an external system.

2.2.4 Spark MLlib

Spark has a very useful library containing common Machine Learning (ML) functionalities, called MLlib. MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionalities such as model evaluation and data import [10]. It is designed to run in parallel on clusters and all its learning algorithms are accessible from all of Spark programming languages. MLib supports a subset of BLAS Standard routines to speed up the processing time.

The BLAS[3] (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing in an optimized and efficient way basic vector and matrix operations. The Level 1 BLAS performs scalar, vector and vector-vector operations, the Level 2 BLAS performs matrix-vector operations, and the Level 3 BLAS performs matrix-matrix operations. Because the BLAS routines are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software.

[3] http://www.netlib.org/blas/

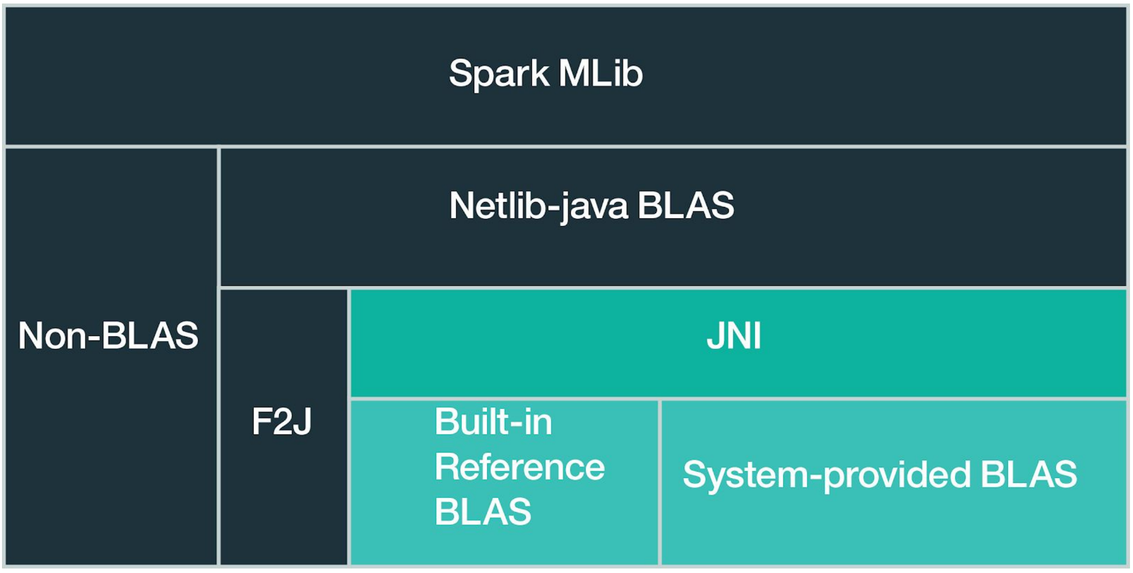


Figure 2 - Spark MLib and BLAS interaction schema

2.2.5 GraphX

GraphX is a library for manipulating graphs and performing graph-parallel computations. It is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs.

3. The Spark Cluster Entities

Apache Spark can run in both *local mode* or *distributed mode* on a cluster. Spark applications run as independent sets of processes on a cluster, all coordinated by a central coordinator. When Spark schedules and runs tasks, it creates a single task for data stored in one partition, and that task will require, by default, a single core in the cluster to execute. Spark offers two ways to tune the degree of parallelism for operations:

- during operations that shuffle data, a degree of parallelism can always be given for the produced RDD as a parameter.
- any existing RDD can be redistributed to have more or fewer partitions.

In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers. The central coordinator is called the **driver**. It communicates with a potentially large number of distributed workers called **executors**. A driver and its executors are together termed a **Spark application**. In the main program of a Spark application (the driver program) there is an object of type SparkContext, whose instance communicates with the Cluster resource manager to require a set of resources (RAM, core, etc.) for executors. This architecture is represented in figure below. Several cluster managers are supported including YARN, Mesos and Spark Standalone Cluster Manager.

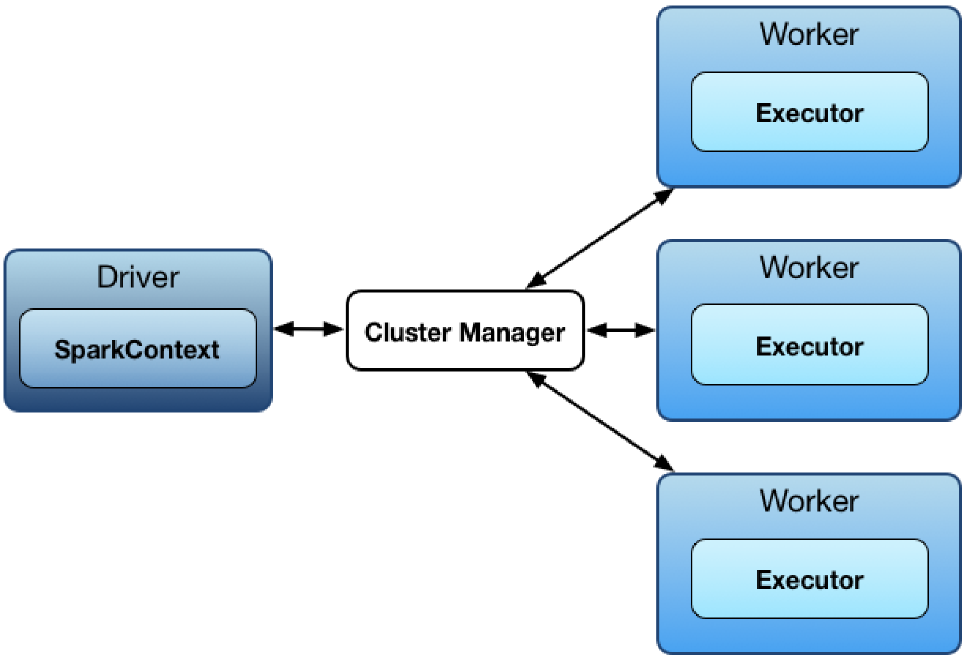


Figure 3 - Distributed execution workflow

The cluster entities are described in detail in the following.

3.1 Driver

The *driver* is the process where the main() method of the program runs. It is the process that executes the user code that creates a SparkContext, creates RDDs, and performs transformations and actions. The Spark driver is responsible for converting a user program into units of physical execution called tasks.

3.2 Executors

Spark *executors* are worker processes responsible for running the individual tasks in a given Spark job. Executors have two roles:

- they run the tasks that make up the application and return results to the driver.
- they provide in-memory storage for RDDs that are cached by user programs, through a service called the Block Manager that lives within each executor.

The cluster manager are described in the next paragraph.

4. Cluster Manager

Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this, while maximizing at the same time the flexibility, Spark can run over a variety of cluster managers, including *Hadoop YARN* (Yet Another Resource Negotiator)[4], *Apache Mesos*[5], and a simple cluster manager included in Spark itself called the *Standalone Scheduler***.**

Hadoop YARN

Hadoop YARN is a distributed computing framework for job scheduling and cluster resource management characterized by High Availability (HA) both for masters and slaves. Running Spark on YARN is useful, because it lets Spark access (Hadoop Distributed File System) HDFS data quickly, on the same nodes where the data is stored. Some YARN clusters are configured to schedule applications into multiple "queues", for resource management purposes. When Spark applications run on a YARN cluster manager, resource management, scheduling, and security are controlled by YARN.

Apache Mesos

Apache Mesos is a general-purpose cluster manager that can run both analytics workloads and long-running services on a cluster. Mesos offers two modes to share resources between executors on the same cluster. In "fine-grained" mode, the executors scale up and down the number of CPUs they claim from Mesos as they execute tasks: in this way a machine running multiple executors can dynamically share CPU resources between them. In "coarse-grained" mode, Spark allocates a fixed number of CPUs to each executor in advance and never release them, until the application ends, even if the executor is not currently running tasks.

Standalone Cluster Manager

If the user want to run Spark by itself on a set of machines, the built-in Standalone Cluster Manager mode is the easiest way to deploy it. Spark Standalone Cluster manager offers a simple way to run applications on a cluster. It consists of a master and multiple workers, each one with a configured amount of memory and CPU cores. When the user submits an application, the user can choose how much memory his executors will use, as well as the total number of cores across all executors. To submit an application to the Standalone Cluster Manager, the user passes cluster URL as the master argument to *spark-submit*. This cluster URL is also shown in the Standalone Cluster Manager Web UI. The Standalone Cluster Manager supports two deploy modes where the driver program of the application runs.

In the Standalone Cluster Manager, resource allocation is controlled by two settings:

- *Executor memory*: each application will have at most one executor on each worker, so this setting controls how much of that worker's memory the application will claim.
- *Maximum number of total cores*: this is the total number of cores used across all executors for an application. The Standalone Cluster Manager works by spreading out each application across the maximum number of executors by default.

4.1 Comparison between cluster managers

In the following, several aspects of Spark Standalone Cluster Manager, Apache Mesos and Hadoop YARN will be discussed, including:

- Cluster Management Scheduling Capabilities;

[4] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[5] http://mesos.apache.org/

- High Availability;
- Security;
- Monitoring.

Cluster Management Scheduling Capabilities

Apache Hadoop YARN has a ResourceManager with two parts, a ***Scheduler*** and an ***Applications Manager***. The *Scheduler* is a pluggable component. Two different implementations are provided, a Capacity Scheduler, useful in a cluster shared by more than one organization, and the FairScheduler. Within a queue, resources are shared between the applications. The *Applications Manager* is responsible for accepting job submissions and starting the application specific *Applications Master*.

Apache Mesos has master and slave processes and allows for fine-grained control of the resources in a system such as CPUs, memory, disks, and ports. Apache Mesos also offers course-grained control of resources where Spark allocates a fixed number of CPUs to each executor in advance which are not released until the application exits.

Spark Standalone uses a simple FIFO scheduler for applications. Each application uses all the available nodes in the cluster. The number of nodes can be limited per application, per user.

High Availability (HA)

Apache Hadoop YARN supports manual recovery using a command line utility and supports automatic recovery.

Apache Mesos cluster manager also supports automatic recovery of the master. Tasks which are currently executing continue to do even in the case of failure.

Spark Standalone cluster manager supports automatic recovery of the master and it also supports manual recovery using the file system.

Security

Hadoop YARN has security for authentication, service level authorization, authentication for Web consoles and data confidentiality.

Mesos provides authentication for any entity interacting with the cluster: the slaves registering with the master, frameworks submitted to the cluster, and operators using endpoints such as HTTP endpoints. Each of these entities can be enabled to use authentication or not.

Spark Standalone supports authentication via a shared secret with all the cluster managers. The standalone manager requires the user to configure each of the nodes with the shared secret.

Monitoring

Hadoop YARN has a Web UI for the ResourceManager and the NodeManager. The ResourceManager UI provides metrics for the cluster, while the NodeManager provides information for each node and the applications and containers running on the node.

Apache Mesos provides various metrics for the master and slave nodes accessible via a URL.

Spark Standalone has a Web UI which permit to view cluster and job statistics as well as detailed log output for each job. If an application has logged events for its lifetime, the Spark Web UI will automatically reconstruct the application's UI after the application exists.

4.2 Pros and Cons of the cluster managers

The primary limitation of Spark Standalone Cluster Manager is its incapability of dynamically adjusting resource usage, because it requires each application to explicit the number of resources that it will consume. Spark Standalone is the simplest of all the considered scheduling frameworks.

YARN offers much more flexible approach to cluster management, because it provides a more robust framework for running a cluster, by allocating new jobs on the physical resources available.

Mesos supports dynamic allocation of resource and elastic scaling of available resources within a cluster. One of the major advantages of Mesos is that it does not depend on Hadoop and it is adaptable to a much wider variety of use cases.

5. PBS-Based HPC Environments Torque

Most high performance computing systems include a batch processing or resource management system as part of their user environment. The HPC architecture has evolved and this has been a fundamental change in the type of data managed in these systems. Managing the proliferation of digital unstructured data in this Big Data gives advantages to high-throughput, high-availability storage system. One of the most commonly used types of HPC batch system is the Portable Batch System (PBS), such as TORQUE [3]. PBS is a batch queueing and workload management system. The principal scope of PBS is to provide controls for initiating and scheduling the execution of batch jobs. It loops through the queued job list and starts any job that fits in the available resources.

HPC clusters are equipped with heterogeneous storage devices and high performance interconnects. As a result, different nodes in the cluster have different storage characteristics. HPC clusters host heterogeneous nodes with different types of storage devices. HPC systems use i) a global parallel file system, with no local storage and ii) dedicated I/O subsystems, where storage is attached to a "centralized" file system controller. Each and all nodes can see the same amount of storage, and bandwidth to storage is carefully provisioned by the system.

HPC models, like TORQUE, have mostly met users' requirements in high computational performance, while Big Data frameworks, such as Spark, have performed likewise in terms of high-level programming, resiliency and I/O handling.

5.1 TORQUE

TORQUE (Terascale Open-source Resource and Queue manager) [3] is a resource management system for submitting and controlling jobs on clusters, that provides control over batch jobs and distributed compute nodes. It manages jobs that users submit to various queues on a computer system and incorporates significant advancements in the areas of scalability, fault-tolerance, usability, functionality, and security development.

TORQUE provides enhancements over other resource managers in the following areas:

- fault tolerance;
- scheduling interface;
- scalability;
- usability.

TORQUE's main roles and purposes are:

- provides job queuing facility;
- monitors resource configuration, utilization, and their state;
- provides remote job execution and job management facilities;
- reports information to cluster scheduler;
- receives directions from cluster scheduler;
- handles user client requests.

5.2 TORQUE Infrastructure

TORQUE batch system consists of three main components:

- **The server**, which is responsible for verifying the integrity and correctness of all requests.
- **The scheduler**, which is responsible for planning the execution of jobs onto computing nodes.
- **The mom daemons**, which run on each computing node and are responsible for reporting the current state of the node, state of jobs running on the node, execution of new jobs and coordination of multi-node jobs life-time.

The TORQUE scheduler has limited functionalities and can be replaced with an external scheduler. In such configuration, TORQUE manages the job requests and the computing resources while the external scheduler queries the server and the jobs running on the computing resources, in order to obtain update jobs and node informations and to schedule jobs in accordance with specified policies, priorities, and reservations.

TORQUE provides a method for handling the jobs that execute simultaneously, based on a first-come first-served basis. In this way, all jobs will run more efficiently and can finish faster, since each one of them is allowed to use all system resources, if available, for the whole duration of its run.

5.3 Yoda HPC cluster hardware details

The HPC Cluster used in experimental assessment, named "*Yoda*", is composed by 30 homogeneous nodes, whose operating system is CentOS 6. Users are shared using LDAP and cached locally using SSSD; home directories are shared from the storage node to the rest of the cluster using nfs3. The resource manager is TORQUE. In details, the hardware of each node is configured as follows:

- Total of 30 nodes.
- 1 login node (name: yoda).
- 1 storage node (name: storage).
- 28 computing nodes named blade[00-09,16-25] and sled[00-07].

The login node is a Dell R620 server equipped with:

- 2 x Intel Xeon CPU E5-2670 @ 2.60GHz 540 8c.

- 192 GB of total memory.
- 2 x 1TB NL-SAS 7.2 krpm HDD.

The storage node is a Dell R720XD server equipped with:

- 2 x Intel Xeon CPU E5-2670 @ 2.60GHz 8c.
- 192 GB of total memory.
- 2 x 500 GB SATA-III 7.2 krpm HDD (for the o.s.).
- 24 x 300 GB SAS-2 15 krpm (for the data).

The storage server is connected to a storage area network that hosts a PowerVault MD3660f System equipped with:

- 2 redundant fibre channel controllers.
- 2 x 200 GB SAS-2 SSD HDD used for cache.
- 135 HDD da 4 TB NL-SAS 7.2 krpm HDD used for data.

The blade nodes are Dell M620 blade servers equipped with:

- 2 x Intel Xeon CPU E5-2670 @ 2.60GHz 8c.
- 192 GB of total memory.
- 2 x 600 GB SAS-2 10 krpm HDD.

The sled nodes are Dell C8220X servers equipped with:

- 2 x Intel Xeon CPU E5-2670 @ 2.60GHz 8c.
- Total Memory 192 GB.
- 2 x 600 GB SAS-2 10 krpm HDD.
- 2 x GPGPU NVIDIA K20.

The cluster has two networks:

- An Infiniband network implemented with 2 Mellanox M4001F, 56 Gb/s switches used by the applications.
- A service 10 Gigabit Ethernet used to share file system data, and for system services communication.

The architecture of the integration Spark with TORQUE is depicted in the next Figure 4.

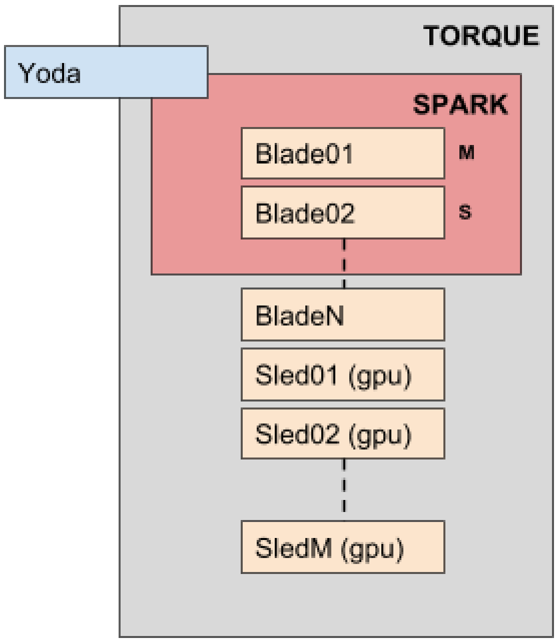


Figure 4 - Spark-TORQUE integration schema

TORQUE's mom daemon run on every Yoda's node: sleds with GPUs and CPUs and blades with CPUs only. In every Spark cluster there is a master node and N slave nodes.

In order to run Spark on TORQUE, a user need to specify a precise number of nodes to use for her computation. For example the code specifies:

```
[@yoda]$ qsub -lnodes=10:ppn=16, walltime=72:00:00, x="NACCESSPOLICY:SINGLEJOB" lanch-spark.sh
```

-
- **`qsub`** It is the basic command for running jobs.
  `--lnodes=10:ppn=16` It reserves 10 nodes with 16 processors.
- **`walltime=72:00:00`** It reserves the nodes for 72 hours.
- `x="`**`NACCESSPOLICY:SINGLEJOB`**`"` It requires the exclusive use of the nodes.
- **`launch-spark.sh`** It runs Spark.

We considered a standalone architecture. At this aim we write the following bash script.

Table 1: Script to launch the Spark cluster: launch-spark.sh

```
#!/bin/bash
if [ -e ~/spark-init.sh ] ; then
    . ~/spark-init.sh
fi

killslave () {
        for h in `uniq $PBS_NODEFILE` ; do
                ssh $h pkill -9 -f org.apache.spark.deploy.worker.Worker
        done
        stop-master.sh
        stop-slave.sh
}
```

```
12. trap killslave SIGINT

13. MASTER=`head -1 $PBS_NODEFILE`;
14. PORT=7077
15. spark_master="spark://$MASTER:$PORT"

16. echo -n "starting spark master on $MASTER... ";
17. $SPARK_HOME/sbin/start-master.sh
18. echo "done";
19. sleep 2;
20. echo "spark cluster web interface: http://$MASTER:8080"  >$HOME/spark-info
21. echo "           spark master URL: spark://$MASTER:7077" >>$HOME/spark-info

22. for n in `uniq $PBS_NODEFILE`; do
23.    echo -n "starting spark slave on $n..."
24.    if [ "$n" == "$MASTER" ]; then
25.       $SPARK_HOME/sbin/start-slave.sh $spark_master
26.    else
27.             ssh  $n  '  .  ~/spark-init.sh;  $SPARK_HOME/sbin/start-slave.sh'  $spark_master  >&
spark-slave-$n.log
28.    fi
29.    echo "done";
30. done

31 wait
```

The first 4 lines of the script in Table 1 are necessary to initialize the environment variables as described in Table 2. With lines 5 to 11 we define a function called killslave to kill all launched nodes (masters and slaves) at the end of the torque job: this is due to the necessity to leave a clean environment. Line 12 specify to execute killslave when the shell receives signal SIGINT. The line's block 13-21 is needed to start the Spark master node and to print on the standard output all the information about its IP address and port number and to save these information in a file named spark-info. Finally, from line 22 to 31, all the slave nodes are started using the PBS variables $PBS_NODEFILE that contains a list of all the Yoda's nodes selected for the job's execution. All the spark nodes are started using an SSH connection and these Yoda's nodes.

Table 2: Initialization of the environment variables needed for the cluster

```
1. export SPARK_HOME=$HOME/spark-2.1.0-bin-custom-spark
2. export JAVA_HOME=/opt/software/java/jdk1.8.0_111
3. if [ -z "$LD_LIBRARY_PATH" ] ; then
4.  export LD_LIBRARY_PATH = /opt/software/gcc/4.8.5/lib64:/opt/software/blas/openblas/lib
5. else
6.
7. LD_LIBRARY_PATH+=:/opt/software/gcc/4.8.5/lib64:/opt/software/blas/openblas/lib
8. fi
9. export PATH=$JAVA_HOME/bin:$SPARK_HOME/sbin:$SPARK_HOME/bin:$PATH
```

Another important aspect is the configuration file $SPARK_HOME/conf/spark-defaults.conf (see Table 3). This configuration file makes it possible to set up all the aspects of the cluster; in details in line 1 must to be indicated the address and the port number of the spark-master; in line 3 and 4 the log options in terms of enabling the logs and where to store them; the 5-6-7 lines are useful to prevent Java errors due to the heap size memory allocation. All the details and options that is possibile to define for the cluster optimization can be found in https://spark.apache.org/docs/latest/configuration.html.

Table 3: Example of spark-defaults.conf

```
spark.master                      spark://sled07:7077
#spark.master                     local[*]
spark.eventLog.enabled            true
spark.eventLog.dir                /home/users/gargiulo/spark_log
spark.serializer                  org.apache.spark.serializer.KryoSerializer
spark.driver.memory               100g
spark.executor.memory             50g
#spark.shuffle.consolidateFiles  true
#spark.shuffle.manager           SORT
#spark.local.dir                 /tmp
#spark.default.parallelism       100
#spark.sql.shuffle.partitions    100

#spark.memory.fraction          0.4
#spark.memory.storageFraction   0.4
#spark.sql.inMemoryColumnarStorage.compressed   true
#spark.sql.inMemoryColumnarStorage.batchSize    10000
 #spark.executor.extraJavaOptions     -XX:+PrintGCDetails  -Dkey=value  -Dnumbers="one  two
three"
```

We encountered a problem with the number of open files at same time, when the files dimension involved into the process increases. In this case, the system automatically creates a huge number of partitions and there is the necessity to open a lot of files. We resolved this issue modifying the `ulimit` size to infinite.

Why to integrate Spark with TORQUE? The integration of cluster-based analytics software such as Spark with an HPC batch environment like TORQUE has a number of potential benefits:

- Spark allows the analytics software to coexist with and leverages the investments made for more traditional HPC applications;
- the integration of Spark, cluster-manager Standalone, allows simplified management of the resource usage accounting and allocation management, which neither YARN nor Mesos currently provides;
- Spark allows for automated workflows that include analytics applications and traditional HPC applications.

The experimental assessment proves all aforementioned advantages, as shown in next Section 6.

6. Experimental Results

In this section we will present a set of tests with the purpose to show the effectiveness of the integration of Spark and TORQUE, demonstrating the scalability of computational time simply incrementing the Spark nodes involved in the computation. We will also show the programming code of these tests to demonstrate the effectiveness of this approach in terms of code-lines.

To this end, we consider the time needed to realize these four types of tests:

- Count of elements in a file: counting of a large dataset of 11,150,089 elements.
- Count of Inner Join: the inner join of a file of 11,150,089 elements with itself.
- Count of Cross Join: a cross join of a file of 33,563 elements obtaining a square number of elements: 1,126,474,969.
- Count of Cross Join BIG: this test is exactly the same of the previous one except for the number of elements involved, in this case 89,129, obtaining a total number of final elements equals to: 7,943,978,641.

This experiments are implemented using very simple Scala code as the one presented in the following. The code evaluates the time needed to count the documents in a collection.

```
val documents = spark.read.parquet("<<docFolder>>")

var res : Long = 0
for (i <- 1 to 10){
        val t1 = System.currentTimeMillis
        documents.count
        val t2 = System.currentTimeMillis
        val time = t2 - t1
        res = res + time
}

res = res/10
println("AverageTime = " + res )
```

The Scala code that realizes the joins (inner and cross) differs from the previous one only for few lines of codes, as shown in detail in the following:

```
#INNER JOIN
val documents = spark.read.parquet("<<docFolder>>")
val documents1 = documents.select(col("pmid").as("pmid1"))

[..]
          documents.join(documents2,col("pmid")===col("pmid1"),"inner").count
[..]


#CROSS JOIN
val documents = spark.read.parquet("<<docFolder>>")
val documents_small = documents.sample(false,0.003)

[..]
          documents_small.crossJoin(Documents_small).count
[..]
```

With the purpose of testing the scalability of the presented solution, we executed the tests on different cluster configurations, in details:

- localhost with 1, 2, 4, 8, 16 cores.
- standalone cluster with: 2, 4, 8 hosts each one with 16 cores.

The following Figures 5 and 6 show the comparison of the performances between the aforementioned different cluster configurations, in terms of computation time for the jobs above described (count, inner join, cross join).

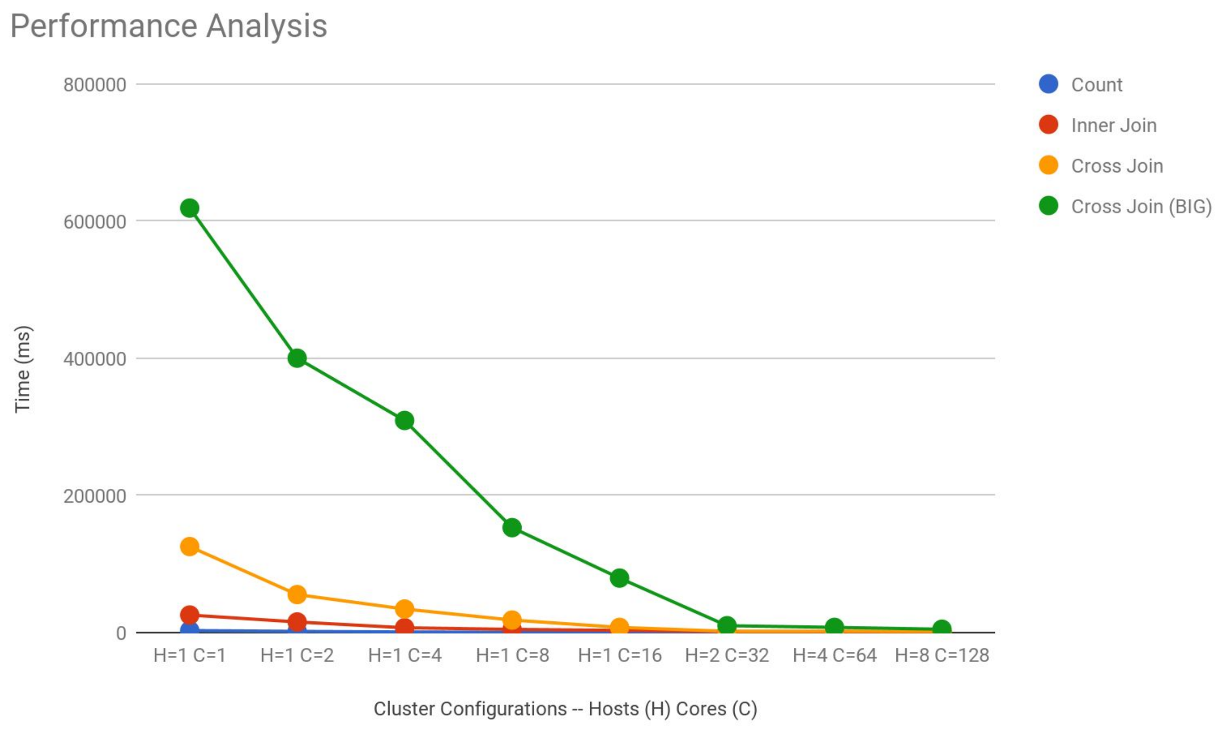


Figure 5 - Performance comparison.

In the previous Figure 5 we can note that there is a performance boost when a Spark cluster with two or more hosts is used. Then, in the next Figure 6 we depict the details of the latter cases, namely a Spark cluster with respectively 2, 4 and 8 nodes. It is worth noting that the mapping strategy increases its own effort when the data involved is big.

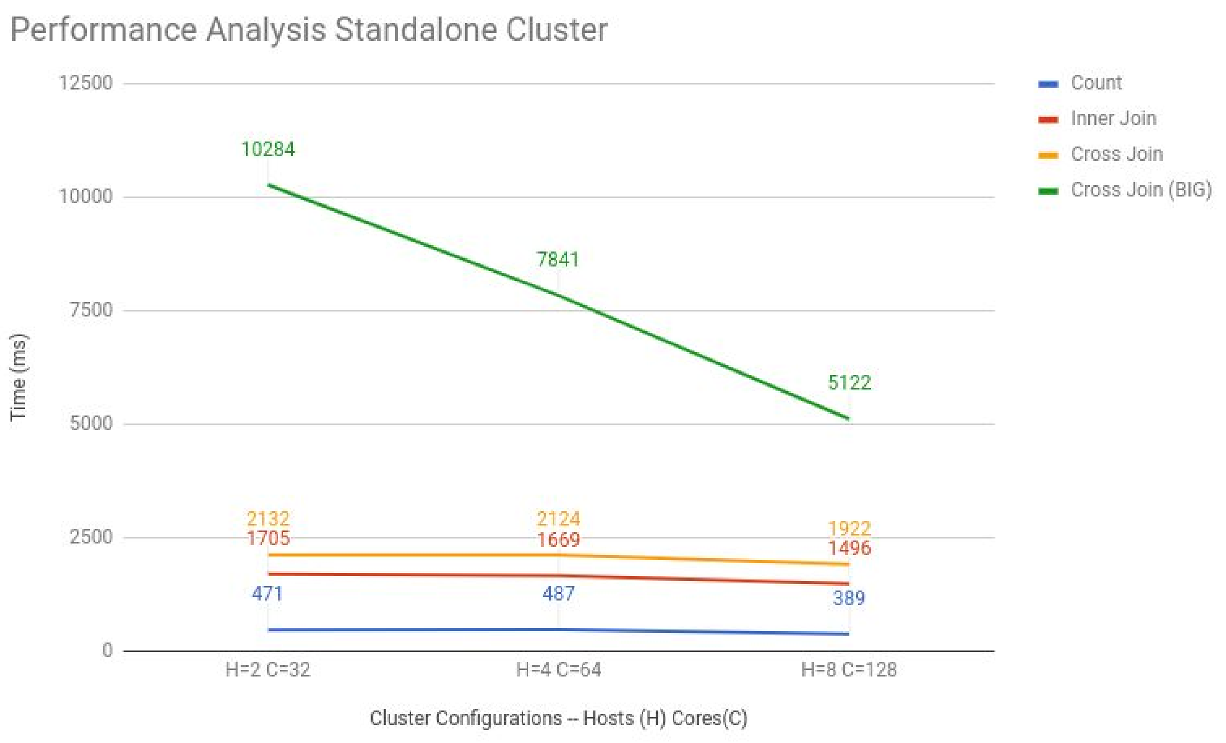


Figure 6 - Standalone Spark clusters performance comparison - two or more hosts case

Another important advantage of Spark programming is the compactness of the programming code. With Scala functional programming approach [11], it is possible to implement in very fast and easy way complex algorithms, exploiting the functional paradigm. We explain the details of the above mentioned advantages through an example. In the following Table 4 we show an example of code comparison, with the same algorithm implemented in Python, with classic imperative programming paradigm, and in Scala, using functional paradigm. We want to demonstrate the compactness and effectiveness of this approach compared to a classical imperative programming approach. It is also worth noting that in the classic script (left side) there are not a parallel programming optimization, which can be obtained only rewriting the code and exploiting a multiprocessing or multi-thread library, but at the cost of a considerable increase of its own complexity. On the other hand, the parallelization is implicit and built-in in the Spark platform: the number of workers involved depends only on the cluster configuration on which the code is deployed.

Looking at the Table 4, we note that the main difference between the two different programming styles is the philosophy behind them. The first one, on the left, is written in a classical procedural/imperative way: in this case, the programmer has to think about a sequential flow of instruction to give in input to the worker. In addition, to implement a parallel version, the programmer has to redefine all the strategies to synchronize the multi-tasks correctly. On the other hand, the Spark program follows a functional philosophy: in this case the programmer has to focus his attention only on the main purpose of the code, in terms of data s/he wants to obtain. As like an SQL script, the programmer has no idea of how the worker will implement the task, and the optimization is demanded to a middleware that will create the various jobs and it will split them upon all the workers in the platform. All the optimizations are demanded to the platform middleware. The result is the very compact code shown in the right side of the table, which requires no effort to obtain a very effective and fast multiprocessing using all the available hosts and cores of the platform.

Table 4: Comparison between procedural and functional code to evaluate the average number of labels (MESH) for document and the average number of documents per label. The documents are identified by a PMID (PubMed Identification Number)

Procedural Single Thread (PYTHON)

```
[import …]

start_time = time.time()

fileCsvMeshInput = sys.argv[1]
fileCsvInput= sys.argv[2]

train = []
labels = []
labelsStat = []
df_data = pandas.read_csv(fileCsvInput,
sep=";")
inputData = df_data.values.tolist()
parole = []
df_mesh = pandas.read_csv(fileCsvMeshInput)
temp = df_mesh.values.tolist()
meshList = []
labelnumber= []

for elem in temp:
    meshList.append(elem[0])

for i in range(len(meshList)):
    labelsStat.append(0)

for elem in inputData:
    text=[]
    parola = 0
    numlabel = 0
    words = elem[2].split(",")
 for i in range(len(words)):
        parola = parola + 1
    parole.append(parola)
    labelMesh = []
    for i in range(len(meshList)):
        labelMesh.append(0)
    for type in elem[1].split("|"):
        labelMesh[meshList.index(type)] = 1
         labelsStat[meshList.index(type)] = 1 +
labelsStat[meshList.index(type)]
        numlabel = numlabel+1
    labelnumber.append(numlabel)
    labels.append(len(elem[1].split("|")))


print("Average words per document "+
str(sum(parole)/len(parole)))
print("Average label per document " +
str(sum(labelnumber)/len(labelnumber)))
print("Average document per label " +
str(sum(labelsStat)/len(labelsStat)))

end_time = time.time()

print("tempo " +str(end_time-start_time))
```

SPARK (SCALA)

```
[import …]

val t1 = System.currentTimeMillis

val dati = spark
  .read
  .format("com.databricks.spark.csv")
  .option("header","true")
  .option("delimiter",";")
  .load(sys.argv[1])

dati.select(col("PMID"),
explode(split(col("MESH"),"\\|")).as("MESH"))
  .groupBy(col("PMID"))
  .agg(count(col("PMID")).as("count"))
  .agg(avg(col("count")))
  .show

dati.select(col("PMID"),
explode(split(col("MESH"),"\\|")).as("MESH"))
  .groupBy(col("MESH"))
  .agg(count(col("MESH")).as("count"))
  .agg(avg(col("count")))
  .show


val t2 = System.currentTimeMillis
val time = t2 - t1

println("Time= " + time )
```

6.1 BLAS Performance Evaluation

To evaluate the increase of performances using the BLAS technology, we implemented the following SCALA code to calculate the cosine distance between two vectors.

Table 5: Code the cosine distance between two vectors calculation

```
import com.github.fommil.netlib.BLAS.{getInstance => blas}
import org.apache.spark.ml.linalg.Vector

 /**
    * interface defining similarity measurement between 2 vectors
    */
 trait VectorDistance extends Serializable {
   def apply(vecA: Vector, vecB: Vector): Double
 }

 /**
    * implementation of [[VectorDistance]] that computes cosine similarity
    * between two vectors
    */
 object Cosine extends VectorDistance {

   def apply(vecA: Vector, vecB: Vector): Double = {
     val v1 = vecA.toArray.map(_.toFloat)
     val v2 = vecB.toArray.map(_.toFloat)
     apply(v1, v2)
   }

   def apply(vecA: Array[Float], vecB: Array[Float]): Double = {
     val n = vecA.length
     val norm1 = blas.snrm2(n, vecA, 1)
     val norm2 = blas.snrm2(n, vecB, 1)
     if (norm1 == 0 || norm2 == 0) return 0.0
     blas.sdot(n, vecA, 1, vecB, 1) / norm1 / norm2
   }
 }
```

The code is used within a SPARK-SQL query as shown below:

```
import org.apache.spark.ml.linalg.{Vector}
import org.apache.spark.sql.{Dataset, Row}
import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.functions._


trait Predictor extends Serializable{
  def apply( trainingSet:Dataset[modelloW2V], testSet:Dataset[Row] ): Dataset[Row]
}

object predictorW2V extends Predictor{


  val cosine = (vecA:Vector,vecB:Vector) => Cosine.apply(vecA,vecB)

  val cosineUDF = udf(cosine)

  override def apply(trainingSet: Dataset[modelloW2V], testSet:Dataset[Row]): Dataset[Row] = {
    val ris = trainingSet.join(testSet,col("rif")===col("riferimento"),"inner")
      .withColumn("cosineSimilarity",cosineUDF(col("vectorsum(vector)"),col("vector")))
      .orderBy(desc("cosineSimilarity"))
     .select(col("pmid"),col("title"),col("abstractText"),col("cosineSimilarity"))
   return ris
  }
}
```

The BLAS technology (we used openBLAS[6] implementation, an optimized BLAS library) can be enabled configuring a system environment variable as follow:

```
export LD_LIBRARY_PATH=/opt/software/gcc/4.8.5/lib64:/opt/software/blas/openblas/lib:$LD_LIBRARY_PATH
```

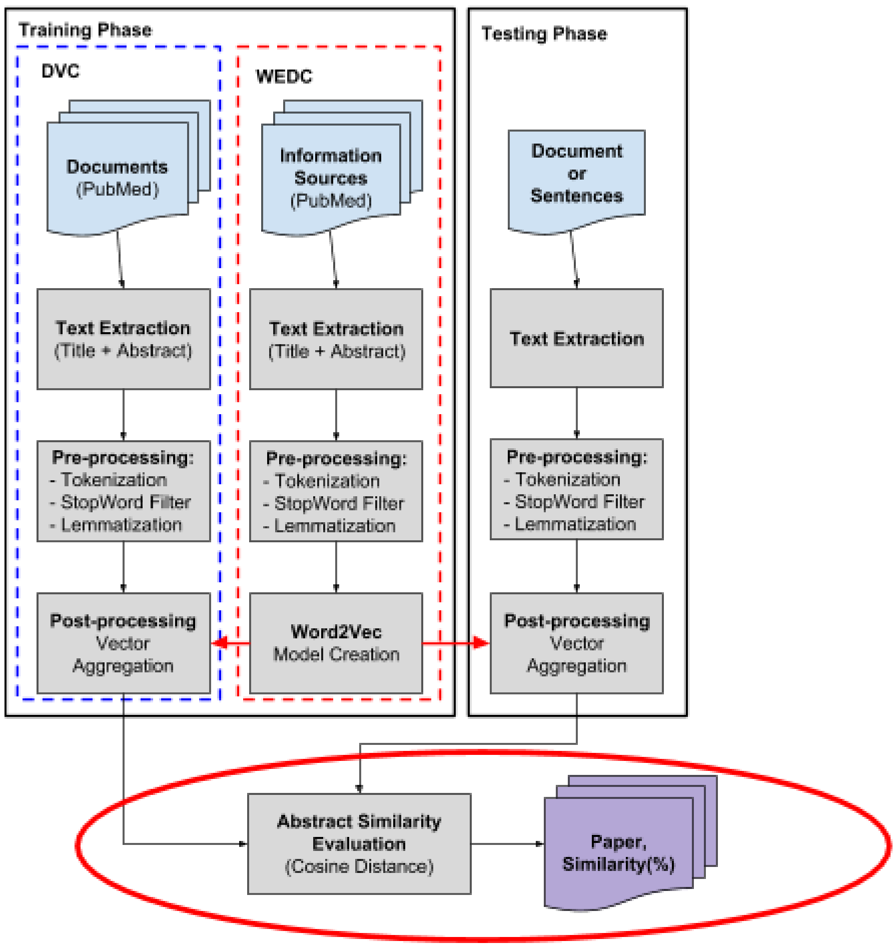
To test if the BLAS correctly works, the following instructions can be used:

```
$export LD_LIBRARY_PATH=/opt/software/gcc/4.8.5/lib64:/opt/software/blas/openblas/lib:$LD_LIBRARY_PATH]
[This previous command gives the possibility to use or not to use the BLAS library]
$spark-shell
scala> import com.github.fommil.netlib.BLAS
scala> println(BLAS.getInstance().getClass().getName())
```

[WITHOUT BLAS CONFIGURED]	**[WITH BLAS CORRECTLY CONFIGURED]**
WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS *WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS* *com.github.fommil.netlib.F2jBLAS*	*com.github.fommil.netlib.NativeSystemBLAS*

With the purpose of evaluating the BLAS impact on the computational time, we considered the scenario depicted in [6], in which the authors proposed a Deep Learning approach to realize a Word Embeddings (WEs) similarity based search tool, considering the medical literature as case study. In the next Figure 7, a graphical overview of the proposed methodology is depicted. The final stage of this architecture must calculate a cosine similarity among a very large number of data. To evaluate the BLAS efforts, we considered the usage of the BLAS calculating 11,150,089 cosine similarities needed in the *Abstract Similarity Evaluation* block.

[6] https://www.openblas.net/



Training Phase
DVC
Documents (PubMed)
Text Extraction (Title + Abstract)
Pre-processing: - Tokenization - StopWord Filter - Lemmatization
Post-processing Vector Aggregation
WEDC
Information Sources (PubMed)
Text Extraction (Title + Abstract)
Pre-processing: - Tokenization - StopWord Filter - Lemmatization
Word2Vec Model Creation
Testing Phase
Document or Sentences
Text Extraction
Pre-processing: - Tokenization - StopWord Filter - Lemmatization
Post-processing Vector Aggregation
Abstract Similarity Evaluation (Cosine Distance)
Paper, Similarity(%)

Figure 7 - The figure represents the conceptual schema of the Deep Learning approach to realize a Word Embeddings (WEs) similarity based search tool proposed in [6]. In the red circle the blocks involved into the BLAS evaluation are highlighted

The following bash command is used to un: the experiment on a Spark environment.

```
spark-submit --master=local[40] --name Pubmed --class pubmedService.pubmedPredictor
PubmedW2V.jar

args[0] = "In this paper we proposed a novel Deep Learning approach to realize a Word
Embeddings (WEs) similarity based search tool, considering the medical literature as case
study. Using the compositional properties of the WEs we defined a methodology to aggregate the
information coming from each word to obtain a vector corresponding to the abstracts of each
PubMed article. Through this paradigm it is possible to capture the semantic content of the
papers and, consequently, to evaluate and rank the similarity among them. The preliminary
results with the proposed approach are obtained analysing a subset of the whole the PubMed
collection. The results correctness has been verified by human domain experts, showing that
the methodology is promising."

args[1] = stop-word-list.txt

args[2] = word2vec_model.txt

args[3] = paper2Vec.parquet
```

In the command there are two classes of parameters. The first group is directly related to the Spark environment, in details:

- **--master=local[40]** means that Spark has to use 40 cores in a single node.
- **--name Pubmed** specifies the name of the Spark process.

- **--class pubmedService.pubmedPredictor** specifies the main class to call within the program **PubmedW2V.jar**.

For seek of clarity, the program arguments, the second group of parameters, are highlighted considering the notation "*args[i] =*".

- **args[0]** is the input text to evaluate.
- **args[1]** represents the stop word list to apply to the input text.
- **args[2]** contains the file address of the word2vec model (549,863 WordVectors).
- **args[3]** is the address of the Parquet directory with all the vectors associated to each of the 11,150,089 papers of the training set.

In the next Figure 8 the result obtained in terms of execution time expressed in ms is represented. We note that the use of the BLAS library improves the performances of more than two times (as expected).

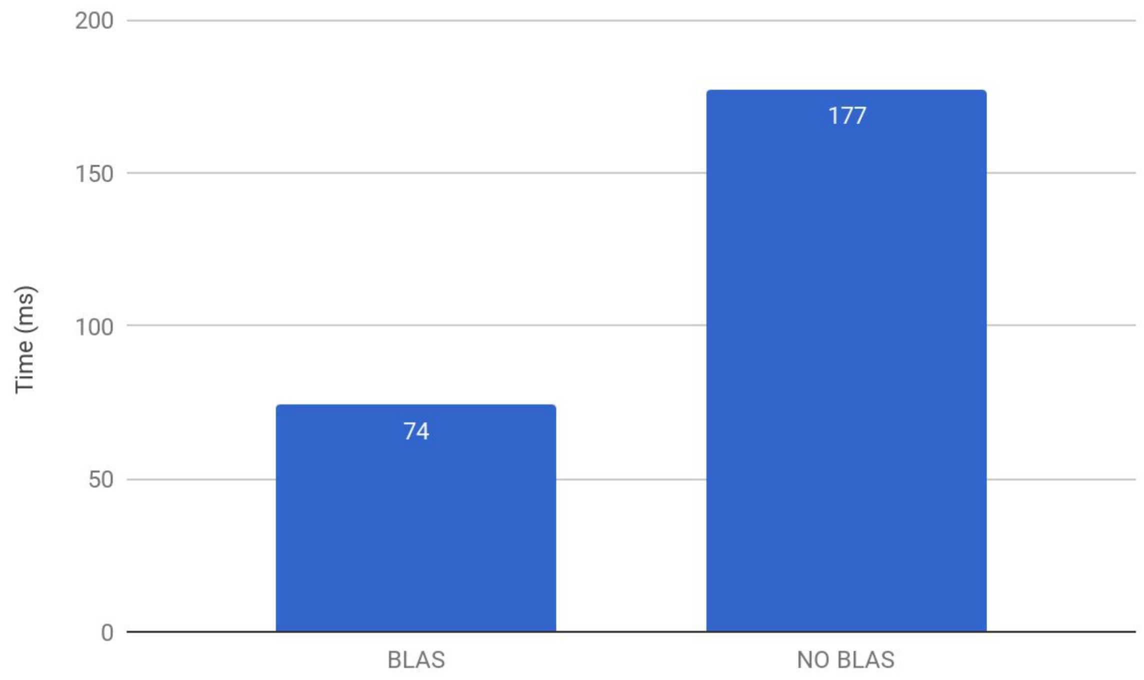


Figure 8 - Histogram of the times in *ms* of the cosine similarity evaluation made using BLAS and NO BLAS technology

7. Conclusion

In this technical report we investigated on the methodology to integrate the Apache Spark Framework into a TORQUE HPC batch environment. We described all details about the installation of this BDA framework on a TORQUE-based cluster, showing all the pro and cons of this process and underlining the compatibility issues. We described all difficulties and we showed how to overcome the most common problems a user can encounter during this process. In addition, we took into account the BLAS library advantages and how to install/configure it properly to correctly run it within this architecture. We also described the main technical details about the use of the aforementioned architecture.

After the installation of Spark on TORQUE cluster environment, we performed an experimental assessment with the purpose of analysing the performances that this kind of system can achieve. Finally, we show some results using a very challenge use case about the realization of a document search engine for the whole

PubMed literature, aiming at the evaluation of the performances of this kind of architecture in a real world application.

The obtained results prove that the use of Spark on a TORQUE HPC Cluster has no impact on scalability and general Spark performances, allowing both technologies to coexist. In this way, the hardware investments of a HPC cluster can be exploited for various tasks and not only for running a Spark environment.

Bibliography

[1] Zaharia, Matei, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11, ACM, 2016.

[2] Xin, R.S., et al., "Shark: SQL and rich analytics at scale." *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013.

[3] TORQUE Resource Manager, http://www.adaptivecomputing.com/products/open-source/torque/

[4] Baer, T., et al., "Integrating Apache Spark into PBS-based HPC environments." *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. ACM, 2015.

[5] Gargiulo, F., Silvestri, S., and Ciampi, M., "A Big Data architecture for knowledge discovery in PubMed articles." *Computers and Communications (ISCC), 2017 IEEE Symposium on*. IEEE, 2017.

[6] Gargiulo, F., Silvestri, S., Fontanella M., Ciampi, M., De Pietro G., "A Deep Learning Approach for Scientific Paper Semantic Ranking." In *International Conference on Intelligent Interactive Multimedia Systems and Services* (pp. 471-481). Springer, Cham, 2017.

[7] Karau, H., Konwinski, A., Wendell, P., and Zaharia, M., "Learning Spark: lightning-fast big data analysis." O'Reilly Media, Inc., 2015.

[8] Zaharia, M., et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, *2012.*

[9] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I., "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters.", *HotCloud 12* (2012): 10-10.

[10] Meng, Xiangrui, et al. "MLib: Machine learning in Apache Spark." *The Journal of Machine Learning Research 17.1* (2016): 1235-1241.

[11] Chiusano, P., and Bjarnason, R., "Functional programming in Scala". Manning, 2015.