Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

SHAPE Project Axyon: a scalable HPC Platform for AI Algorithms in Finance

Fabio Franzoso[a], Riccardo Folloni[a*] , Jacopo Credi[a], Eric Pascolo[b]

[a]*Axyon AI S.R.L, Italy*
[b]*CINECA Consorzio Interuniversitario, Italy*

Abstract

With the advent of Big Data, the increasing use of AI algorithms in finance and the consequently fast-growing need for computational power, it has become necessary for FinTech companies to use large scalable HPC systems able to handle and explore a wide array of Machine Learning models (e.g. deep neural networks) and to remain up-to-date with state-of-the-art methods.

With this work, we wanted to maximize the efficiency of accessing different types of remote computational resources potentially available to our proprietary Machine Learning platform, without losing the flexibility provided by in-house compute power. This is a mandatory requirement for a FinTech company oftentimes working with proprietary data that cannot be uploaded to cloud systems.

In this whitepaper, we outline the development of a scalable and flexible DB-centric Master-Slave system architecture able to exploit any connected internal or external computational resource (including an HPC cluster) in a flawless and secure way.

Introduction

The adoption of AI in finance [1] has recently seen a fast growth due to the availability of increasing computational performance and the rise of Big Data. The application of Deep Learning (DL) models to time-series forecasting and modelling has been studied for decades, though only recently they have proved themselves useful and were successfully exploited for their predictive and explanatory power in many scientific and business sectors.

One of the main reasons for this late development is that an extremely high computational power (e.g. an HPC cluster) is required in order to train DL models. The other crucial ingredient in this success was the possibility to access huge amounts of data, which became cost-effective due to technological advancements in data collection and storage fields. This has generated the opportunity to train Deep Neural Networks models with Petabytes of new information, thus increasing the accuracy of predictive models. Rather than being a minor incremental improvement, this new possibility may be game-changing for many FinTech startups working on areas in the financial industry where even a marginal improvement in model accuracy may translate into enormous value for their customers, being them individuals, companies or financial institutions.

Given the extremely large amount of data to process with algorithms that often require a lot of computational power, it may be challenging for a startup to make up for the lack of in-house computational capability and it may be necessary to rely (at least in part) on external cloud computational resources. At the same time, clients in the financial sector often require to keep their data on their premises, as the risk of depending on the security of a third party infrastructure may be too high, or because imposed to do so by regulators. For all the above reasons, having in place a flexible system able to run computational jobs on different types of connected nodes and HPC systems would constitute a massive competitive advantage for a FinTech handling sensitive financial data.

* Corresponding authors e-mail addresses: fabio.franzoso, riccardo.folloni, jacopo.credi@axyon.ai, eric.pascolo@cineca.it

With this work, we aim to reshape our internal machine learning development platform (henceforth also called "Axyon Platform") in order to spawn and manage jobs across several computational resources in an optimized manner and designed to be flexible, scalable and resource-efficient.

About Axyon AI

Axyon AI is a leading player in Deep Learning, the newest area of machine learning, for time-series forecasting. Our superior technology is packaged into tailor-made solutions for capital markets & asset management. The team brings together deep technological expertise with domain knowledge of the financial sector, accrued through a 10-year long experience in software development and a 5-year long research and development effort in artificial intelligence applied to finance. Starting from the development of an automated trading system based on genetic algorithms, we moved into Deep Learning early on, betting on it to become the leading technology in Big Data-based predictive systems. Our company has now experience (and successful products and solutions) in several financial use-cases, starting from credit risk to investment banking, from trading to asset allocation. Existing clients include ING and Nikko Global Wrap Ltd.

Environment

As a test bench to develop a reshaped version of our Axyon Platform, we used the D.A.V.I.D.E.[2] cluster installed in CINECA, the main Italian supercomputing center, as a High-Performance Computer to run jobs on, and an OpenPower9 node available for Axyon to build a container to be exported to other Power9 architectures.

In the following sections, we describe the hardware and software technology used to test and deploy the Platform.

Hardware

- D.A.V.I.D.E. (Development of an Added Value Infrastructure Designed in Europe) is an Energy Aware Petaflops Class High-Performance Cluster based on Power Architecture and coupled with NVIDIA Tesla Pascal GPUs with NVLink. The cluster is composed of 45 nodes (2 Power8 + 4Tesla P100) + 2 (service & login nodes). Each node is equipped with OpenPower8 (8 cores per processor; 16 cores per node) + NVIDIA Tesla P100 SXM2, the internal networks are 2xInfiniBand Extended Data Rate (EDR) at 100 Gb/s, 2x1GbE and total Peak Performance of D.A.V.I.D.E. is 1 PFlop/s (double precision).
- OpenPower9 node: is a single node available at Axyon equipped by OpenPower9 processor (20 cores per processor, 40 cores per node) coupled 4xNVIDIA Tesla V100 SXM2-32GB with NVLink, the Peak Performance of the node is 28 TFlop/s (double precision).

Software

To simplify the installation process, we chose to make heavy use of container technology. On HPC clusters, the Hobson's choice is Singularity technology[2][4], that eliminates all security problems caused by a possible exploit of root permission on the host by having a container running in the user space and allows to expose GPU drivers into a container (see also discussion in [5]). The Singularity version used on the D.A.V.I.D.E. cluster is 3.0.2, while the OpenPower9 node uses version 2.5.2. The seamless compatibility between these two different releases shows the resiliency and the stability of the selected technology. Within the container, our Platform is based on the following open-source deep learning libraries: Theano 1.0.3 and Keras 2.0.8. In order to leverage the full capabilities of the compute nodes, we need to enable NVIDIA GPUs using specific architecture libraries: cuDNN 7.6 based on CUDA 10.0. The entire application has been written using Python 3.6, that is compatible with all the previously mentioned libraries.

Axyon Platform

In this section, we describe the initial structure of the Axyon Platform and the basic working scheme. It is worth noting that in its first prototypal version the Axyon Platform was born as a standalone project whose engine had been entirely designed in PHP and Python 2.7 programming languages. The main components of the legacy architecture are as follows:

- The Axyon Database (ADB): the component on which all jobs' specs are stored and all the information about jobs' execution is kept.
- The Axyon Web Panel (AWP): the component used for job management.

- The Engine: the main component of the project that manages jobs stored on the aforementioned Database and determines which jobs to execute. The engine contains all the scheduling logic, the deep learning libraries interaction for data processing, training, inference, and deployment of machine learning models.
- The Storage: each node has an attached storage from which it can load datasets, store job logs and other useful information, such as performance metrics and informative plots. The storage can be shared (as in the case of the nodes in the Axyon headquarters) or it can be exclusive of one node. The information of where datasets are stored is specified in the configuration file of the Engine Project belonging to the host.

The Panel component and the Database are installed on a single, central machine, whereas the Engine component must be replicated for each machine that has to run jobs locally. Each one of the nodes has a different configuration file in order to execute jobs.

Jobs are managed in a fully decentralized fashion, where each machine with the installed engine has a view of the list of queued jobs, ready to be executed, and determines if it is able to run the job based on its characteristics (priority, node reservation, estimated memory requirements, etc.).

The part of the Engine used to check jobs on database is organized as an ensemble of scripts. A cronjob installed on the host machine is used to run the script every minute in order to check and run queued jobs on DB, if the resources on the host machine are available.

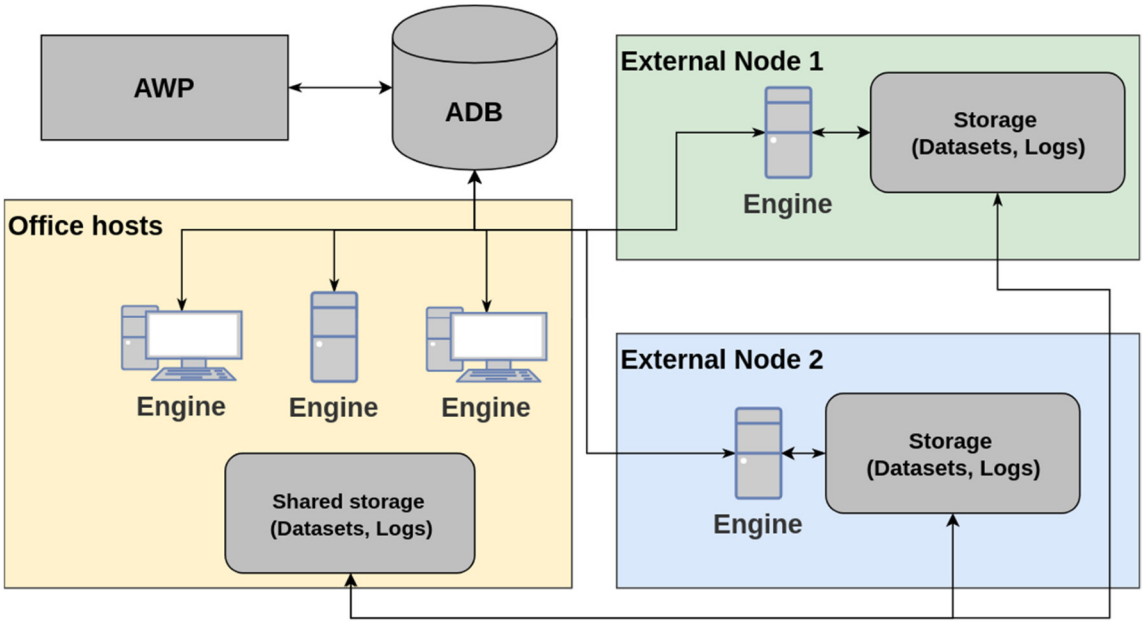

Figure 1: Axyon Platform Infrastructure

This structure has several benefits: the Panel lets us manage jobs through a graphical UI and maintain training metrics, neural network parameters and store all the useful information in an organized fashion. The central Database is accessible from both the Web UI and all the connected engines to manage and execute Jobs. Finally, the Engine can potentially be installed on every machine running a compatible Linux distribution (e.g. a desktop workstation) and quickly transform it into a mini-host able to run jobs.

However, this structure also has some drawbacks. First of all, the engine is not installable as a container, and every time the component has to be placed on a new machine, it is necessary to install the Operating System, all the required programming languages, libraries and Python packages in the correct version. This step requires a lot of manual work each time we wish to connect a new host. Secondly, the storage where datasets are preserved and logs are stored must be continuously kept up to date with the latest generated datasets, whereas logs must be synchronized to the ADB central storage accessed by the AWP. Moreover, each engine node requires persistence on the host and works both as a self-scheduler and a job executor. The node persistency is often not supported by systems composed of a cluster of nodes like HPC systems and adapting the current Engine to run on an HPC system would require a lot of work and resources. Lastly, the scheduling algorithm is decentralized. As a result, the Platform is often unable to manage all available computational resources in an optimal way, which may result in slower execution time.

Work Description

Axyon Platform re-shape

The goal of this work was to build a flexible and scalable Platform in order to extend the compute capability and scalability of Axyon AI. The new infrastructure has been designed to maximize the strengths of different systems of interest for our computing workload:

- In-house infrastructure: easier management of small workloads and test purposes.
- Cloud services: flexible and powerful enterprise hardware for intensive computing tasks.
- HPC: scalable workload requiring several nodes and massively high parallelism.

All the aforementioned infrastructures will be managed using the same programming strategy:

the paradigm implemented in the new *Axyon Workload Manager* (AWM) is based on a DB-centric Master-Slave approach. The *Axyon Database (*ADB) stores all the information needed to the Master-Slave counterpart to manage the linked resources and computational Jobs. Information can be inserted into the ADB using an AWP, in order to manage jobs specifications and results in a fast and visual user interaction.

The Engine of the legacy Axyon Platform was designed to handle a simpler infrastructure. In order to satisfy all discussed requirements, the Engine architecture had to be modified so that the tasks managed by the Engine are split into several new components of the re-shaped architecture (Figure 3).

- The Master part of the AWM is composed of three parts that handle various aspects of a job's life cycle, spanning from the assignment and launch of the job to the end of the computation and result gathering.
- The component of the architecture that first interacts with a job is the *Job supervisor* (AWM-JS) and, specifically, the part that handles the scheduling of the jobs stored in the ADB manipulated through the AWP. The part that collects the results is implemented using *APIs* (AWM-API) and will be described later, in order to explain the architecture functionality by following the workflow of a job. The two elements have been designed to operate independently of each other but logically combined under the Master component.
- To handle the connection between the Master and a slave a new component called *Cluster-Manager* (AWM-CM) has been designed.
- The component of the slave that takes care of providing standardized access to the AWM-CM is called *Proxy* (AWM-PX), which contains a machine-specific module called *Interpreter* (AWM-INT); the latter interacts with the local environment and executes the actions as ordered by the AWM-JS.
- One of the main tasks of the Engine is the execution of AI algorithms. In the reshaped architecture, this task is assigned to the *Axyon Job Runner* (AJR), which is integrated into the container that AWM-PX downloads and launches on the computational resources. During the computational process, the AJR is the one expected to report the progress of the assigned Jobs to the master using AWM-API.

```
Axyon Workload Manager (AWM)

   - Master:
        - API (Flask) → AWM-API
        - Jobs Scheduler → AWM-JS
        - Cluster-Manager → AWM-CM
   - Slave
        - Proxy → AWM-PX
        - Interpreter → AWM-INT

   - Axyon Database → ADB
   - Axyon Web Panel → AWP
   - Axyon Job Runner → AJR
```

Figure 2: AWM schema

Master

The purpose of the AWM-JS is to coordinate the actions requested by the developers, such as creating a new job, stop it, restart the execution or kill it, by abstracting the real computational resources available both locally or remotely. Starting from those ideas and taking into account that resources could be provided by nodes with different architectures, systems or even through a different internal handling logic, the scheduler uses the AWM-CM, which is specific for each computational resource at our disposal. It is responsible for providing a set of actions standardized for the AWM-JS and to translate these actions into calls that the requested cluster can handle, for example by simply building an SSH call to the AWM-PX. Furthermore, the AWM-JS needs to assure that jobs assigned to Slaves are within a time parameter that lets it know whether a job has to be considered alive and act accordingly, maintaining absolute control on who was assigned to execute a job, which jobs are running and which should be marked as dead.

During the scheduling cycle, the AWM-JS synchronizes its internal job queue with the database information and identifies any changes made by users, such as the creation of a new job or the request to kill an already assigned job. Given this information, the AWM-JS is able to assign the respective AWM-PX the correct operation to be executed through the AWM-CM.

AWM-API allows to regularly receive updates from the AJR on the jobs' progress, collecting partial and final results into the ADB. Moreover, it enables the AWM-PX to check and download the latest versions of the datasets and the supported containers. AWM-API uses an encrypted protocol to make the communication secure.

Slave

Moving to the part that executes the operations, we decided to use a slave system with standardized calls, divided into two components: the AWM-JS, through the established communication system handled by the AWM-CM, uses a set of generalized calls available on the AWM-PX to handle the operations required to execute a job.

For each type of cluster identified, an appropriate AWM-INT slave component has been developed. The interpreter deals with the adaptation from the AWM-CM to a more specific set of commands necessary to handle the job. Those commands, as in the HPC case, might have to interface with a scheduler like SLURM [6] in order to handle all the necessary actions.

In addition to the specific AWM-INT, which operates as a macro subdivision of the types of hosts on which our software may be executed, a configuration is available to increase the granularity without losing generalization, granting us the capability to use the same Interpreter for similar hosts.

Container: the core of the platform

The AJR does not run directly inside the computational resource. Instead, it has been designed to be integrated into a containerization software, like Singularity for HPC. The use of a container provides the advantages of an easily replicable tested environment with all the necessary libraries and configurations for the AJR, granting quick access to the computational resources while limiting the initial setup overhead and any additional maintenance cost posed by the addition of a new environment.

When a job needs to be run, a container including the AJR is instantiated. Apart from the input data and the results, which are stored in the host machine, all the rest is bound to the container life, which allows to perform a clean run every time.

Implementation and issues

AWM-JS Main scheduling

The main scheduling cycle focuses primarily on managing the queue of jobs and dispatching them to the correct clusters of nodes connected to the system, while keeping track of available resources on all connected nodes and currently running jobs.

After initialization, the scheduling cycle can be summarized as follows:

1. Load jobs from the ADB
    a. If the internal queue is empty, fill it with all the available information
    b. If the internal queue is not empty and some actions have been executed on the queued jobs, synchronize the ADB with job statuses from the internal queue.
2. Collect/derive an estimation of available resources for each connected AWM-PX.
3. Manage the internal queue and assign requests:
    a. Kill a job
    b. Retrieve which node is executing a job
    c. Submit a job to a resource
4. Execute requested actions on the selected clusters
    a. Execute kill
    b. Execute job-node information retrieval
    c. Assign jobs to the selected AWM-PX
5. Retrieve all job information and update the internal queue
6. Mark all the updated nodes to be updated on the ADB

One of the most important objectives of the AWM-JS is to implement all the functions that manage the job queue. It defines job properties and handles the synchronization with the Database, keeping up to date the ADB information and the internal queue.

1. After the initialization, the main scheduling cycle begins. The first action that the JS has to perform is updating the internal queue of jobs with the information retrieved from the ADB. During the first iteration, the queue of the AWM-JS will be empty and the list of tasks to be executed will be stored on the ADB. Consequently, the JS will have to fill its internal queue using this information.
2. Once the JS has filled its internal queue, it updates the information regarding the connected cluster managers and their available resources, to manage the jobs that will later be assigned.
3. The AWM-JS manages the queue, ordering jobs by their priority, as assigned by the users through the Web UI. The type of actions that the scheduler is able to assign for each job are the following
    a. **Submit**: assign the job to a cluster.
    b. **Kill**: the job has already been submitted on a cluster, but a user has requested to kill it, so the has to be stopped on the cluster.
    c. **Get Job Node**: the job has already been submitted on a cluster and a user wishes to retrieve information about its status, therefore the specific node where it is being executed needs to be retrieved, in order to report this information to the user. This action is also used when a job is presumably running on a cluster but no updates were received within a certain time. If the AWM-JS cannot retrieve any information about the node on which the job is running, then the job will be marked appropriately and in the next iteration it will be checked if it needs to be reassigned.
4. The fourth step of the cycle performs the execution of all the actions previously assigned. There is no overlap between the assigned actions, thus it is possible to execute all of them in a completely asynchronous manner. The final results are then collected in order to begin the next iteration of the cycle. During this phase, it will be established which jobs have to be updated from the AMW-JS queue to the ADB. At the beginning of the following iteration, all the necessary information will be transferred to the Database.

AWM-PX Connection

The connection between the AWM-CM and remote resources is provided by the SSH protocol, that permits to remotely login on the cluster and to launch the AWM-PX software to control the workflow and return the information about a job's status through STDOUT and STDERR. The SSH connection is performed by the *Paramiko* library [7] and configured to use an RSA SSH key. The AWM-CM executes the command in box 1.

```
source $HOME/slave-proxy/slaveInterpreter/bin/setup.sh;
axyon_proxy --token  --config '{"action": "submit", "scheduler":
{"command": "”}}’
```

Box 1: AWM-PX command

First, the AWM-CM calls setup.sh to set the correct environment. This small bash script sets up all environment variables, aliases, path, pythonpath and loads modules. The only path that Master needs to know is the setup script path because all the rest will be configured within the cluster while running the script. Note that the directory $AXYON_DATA is the root path containing all the files for the execution of the AWM-JR, including datasets, logs and output coming from the training of AI algorithms, along with the image of the container to be executed on HPC nodes. The D.A.V.I.D.E. version of setup.sh is reported in box 2. Python and Python related modules are the only external software needed to the axyon_proxy script and are loaded by the module environment provided by sysadmin.

```
# found itself and set AXYON env var
BINDIR=$(cd "$(dirname "${BASH_SOURCE[0]}")"; pwd -P)
export AXYON_HOME=${BINDIR%%slaveInterpreter/bin}
export AXYON_INTERPRETER_HOME=${BINDIR%%/bin}
#set bin path as system path, axyon dir path as pythonpath
export PATH="$AXYON_HOME/bin":$PATH
export PYTHONPATH=$AXYON_HOME:$PYTHONPATH
#set directory local path
export AXYON_DATA=$CINECA_SCRATCH
export HOME_CONTAINER=/home/jr-user
#set alias
alias pyccleandir='find . -name '*.pyc' -delete'
alias axyoncleandir='cd $AXYON_HOME ; pyccleandir ; cd $OLDPWD'
#loads modules
module load python py-pyyaml py-request singularity
```

Box 2: setup.sh script example

As mentioned above, axyon_proxy is called using SSH as a bash script, not as a service. After performing all the tasks requested from the Master, the proxy writes on STDOUT the information in JSON format and exits. To ensure a secure and correct message transmission between Master and Proxy we use a token to encapsulate the final JSON. The AWM-CM, using a regular expression, extracts the message between the two tokens (the same token is used to start and end the communication message). This encapsulation mechanism is necessary because STDOUT and STDERR could potentially also contain other warnings or errors occurred during the execution of the Proxy.

AWM-INT : SLURM interaction

The main task of the Axyon Proxy is the interaction with the HPC schedulers. In the case of D.A.V.I.D.E. the scheduler used is SLURM, the most popular scheduler among supercomputers. The possible actions are described in the “Main scheduling” Section, whereas here it is reported how the interface runs with SLURM.

Submitting a job

The user specifies the resources requested by the job in the AWP, encapsulating this information in a JSON string passed to the AWM-PX that calls the AWM-INT. The latter uses a job template file (box 3) to produce a correctly formatted user request, along with the command, then the new file is saved in `$AXYON_DATA/workdir/job_dir` and submitted with the SLURM command **`sbatch`.**

```
#!/bin/bash

#SBATCH -t $time
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=$cpus
#SBATCH --gres=gpu:$gpus
#SBATCH --partition=$partition
#SBATCH --mem=$mem
#SBATCH --out=$job_stdout
#SBATCH --err=$job_stderr
#SBATCH --account=$account

$command
```

Box 3: Job template

Killing a job

When a kill action is requested by the user, the information about SLURM job number is communicated to the AWM-PX. Then, the AWM-INT checks if the job is still running on one of the connected nodes, retrieving the job list using the following **`squeue`** command, if the job to be killed is found in the retrieved list, the following **`scancel`** command is called (box 4).

```
squeue -h -o%A -u user

scancel job_number
```

Box 4: Kill job commands

Getting job info

If the AWM-PX requests job information, the AWM-INT uses the **`squeue`** command to get it. Choosing different formats of `squeue` strings, we can get the information we need. For example, given a user job, the command below reports the number, the status (run or pending) and the list of nodes allocated to the job. The string will be always in the same format and consequently easy to parse.

```
squeue  -o%12A%5t%20N%
201615344     R    node176
```

Box 5: Example of squeue command with format

Container management and AWM-JR

The developed Singularity images are based on an official Nvidia Docker image [8], which includes CUDA and cuDNN libraries already installed and configured to work inside a container with access to the hardware resources of the host machine. Multiple recipes are maintained due to different architectures that can be encountered. For

D.A.V.I.D.E. and PowerPC-based resources, the base image is nvidia/cuda-ppc64le:10.0-cudnn7-devel-ubuntu18.04 and the minimal GPU driver requirement for hosting machines is version 410.48 or newer.

Our Singularity was designed to limit its size in order to speed up its transfer and updating. Therefore, the image is created including only the necessary software, while folders containing input and output data are mapped to a set of directories on the host.

The setup described above solves several issues encountered during the design and prototyping phases:

- **Size**: datasets are individually small (i.e. a few gigabytes), but quite numerous, thus a generic Singularity image with integrated datasets would increase in size very quickly, in the order of hundreds of gigabytes. Moreover, a single container version would be defined by the combination of software version and the datasets included.
- **Transfer**: Singularity runs on user space, however it needs root privileges for the building phase. On HPC systems, root access is generally not an option and usually contacting the system administrators is needed in order to request a build. This operation can become cumbersome for both sides in the case of development or test versions, which can be considered unstable. Therefore, during the build phase it is necessary to have a compatible architecture with the required privileges and then transfer the result, which makes the process subject to any bandwidth limits present on the building host.
- **Customers data scope**: in the financial sector, as in many other domains (e.g. healthcare), customers are rather careful about where their sensitive data reside and situations where customer data cannot leave their premises are far from uncommon. By splitting the software from the data, it is possible to meet the customers' needs without having to create custom container versions, which would increase maintenance drag.

The first two problems were encountered in the early versions of our Singularity container, where for preliminary tests it was decided to use the "writable" mode during the build phase, thus creating a "permanent" image. This causes a considerable increase in the space used, with the image quickly growing to over 25GB (with no datasets included), compared to the current ~2GB image with host directories mapping. Furthermore, having moved the I/O outside of the Singularity image makes the life cycle of the compute part independent of the life cycle of the data.

The Container launch

The command reported in box 6 is composed by the AWM-JS and substituted in a SLURM job template by the AWM-INT. The Singularity action **exec** is called, which results in the container singularity-ppc64-*_*_*_*.img starting on the compute node and the file to_execute.sh, containing the instruction to run an AWM-JR, to be launched. In order to correctly perform the launch phase of the container, we must pay attention to the path of the files that AWM-JR uses. Inside the container environment, all relative paths must refer to the container's root directory. For this reason, the directories are included in the bind command (using the -B flag) to map D.A.V.I.D.E parallel filesystem directories and the folders inside the container. The flag --nv in the command enables the use of NVIDIA drivers by the container, which grants the use of GPUs by the software installed inside the Singularity image.

```
singularity exec --nv
-B $AXYON_DATA/sources/:$HOME_CONTAINER/sources/
-B $AXYON_DATA/logs/:$HOME_CONTAINER/job-runner/logs/
-B $AXYON_DATA/temp/:$HOME_CONTAINER/job-runner/temp/
-B $CINECA_SCRATCH/axyon_workdir/139737:$HOME_CONTAINER/axyon_workdir/
$AXYON_DATA/container/singularity-ppc64-2_0_0_2.img
$HOME_CONTAINER/axyon_workdir/to_execute.sh
```

Box 6: Singularity command example

AWM-API

The use of HTTPS allows authentication and the secure exchange of data between the parts, encrypted via TLS, without giving direct access to the internal storage where the datasets and the results reside, but limiting the possible actions to a defined set, within their scope and limits.

On the technical side, the API management was implemented using Flask, a Python web micro-framework which, given its peculiarity of being minimal, allows it to be easily tailored according to our needs. In fact, it guarantees good flexibility on how to structure and use it, as opposed to more complete but also rigid frameworks. Moreover, it allows us to change the type of database used without particular problems in the future.

This architecture has also provided the opportunity to solve a problem not strictly related to the development of the new infrastructure but which sometimes has slowed us down, i.e. how to give developers access to datasets in a simple and secure way.

Final workflow

In this section, we report a full working cycle of job assignment and metrics report from D.A.V.I.D.E., as the test bench remote HPC system offered by CINECA supercomputing centre for these trials.

1. The user, using the AWP, inserts a Job on the ADB.
2. The AMW-JS, in the synchronization phase between the scheduler queue and ADB queue, detects that a new job is present on the ADB and has to be assigned to one of the connected AWM-PX.
3. The AWM-JS retrieves the necessary information about the connected clusters and manages the scheduler queue, assigning the previously inserted Job to one of the AWM-PX. For example, let D.A.V.I.D.E. be the selected cluster.
4. During the action execution phase, the AWM-JS executes the action of job assignment to D.A.V.I.D.E. using the appropriate AWM-CM. At this step, there is a communication phase between the AWM-PX and the AWM-CM. The AWM-PX, using the AWM-INT interface, checks if its container version is correct and if it possesses the dataset required by the job. If not, it downloads them and prepares for the job execution.
5. The job is sent to the SLURM queue on D.A.V.I.D.E., waiting to become a running instance across the cluster nodes.
6. The result of the assignment is communicated to the AWM-CM.
7. If the execution assignment is successful, the job is marked as "Assigned to D.A.V.I.D.E." on the ADB. On the other hand, if something goes wrong during the assignment, the job remains in the queue along with a warning message.
8. The job is now running on D.A.V.I.D.E.'s nodes, inside a container with the AJR installed. Nodes belonging to the cluster can read from a *parallel file system* internal to the cluster and retrieve the dataset previously synchronized. The instance can communicate with external hosts and in particular it reports the job results to the AWM-API connected to the Master. The AWM-API is used for metrics and outputs communication to store information on the ADB.
9. At the end of the execution of a job, the instance will send the final message to the Master through the AWM-API, marking the Job as "Completed" and ready for the user to analyse.

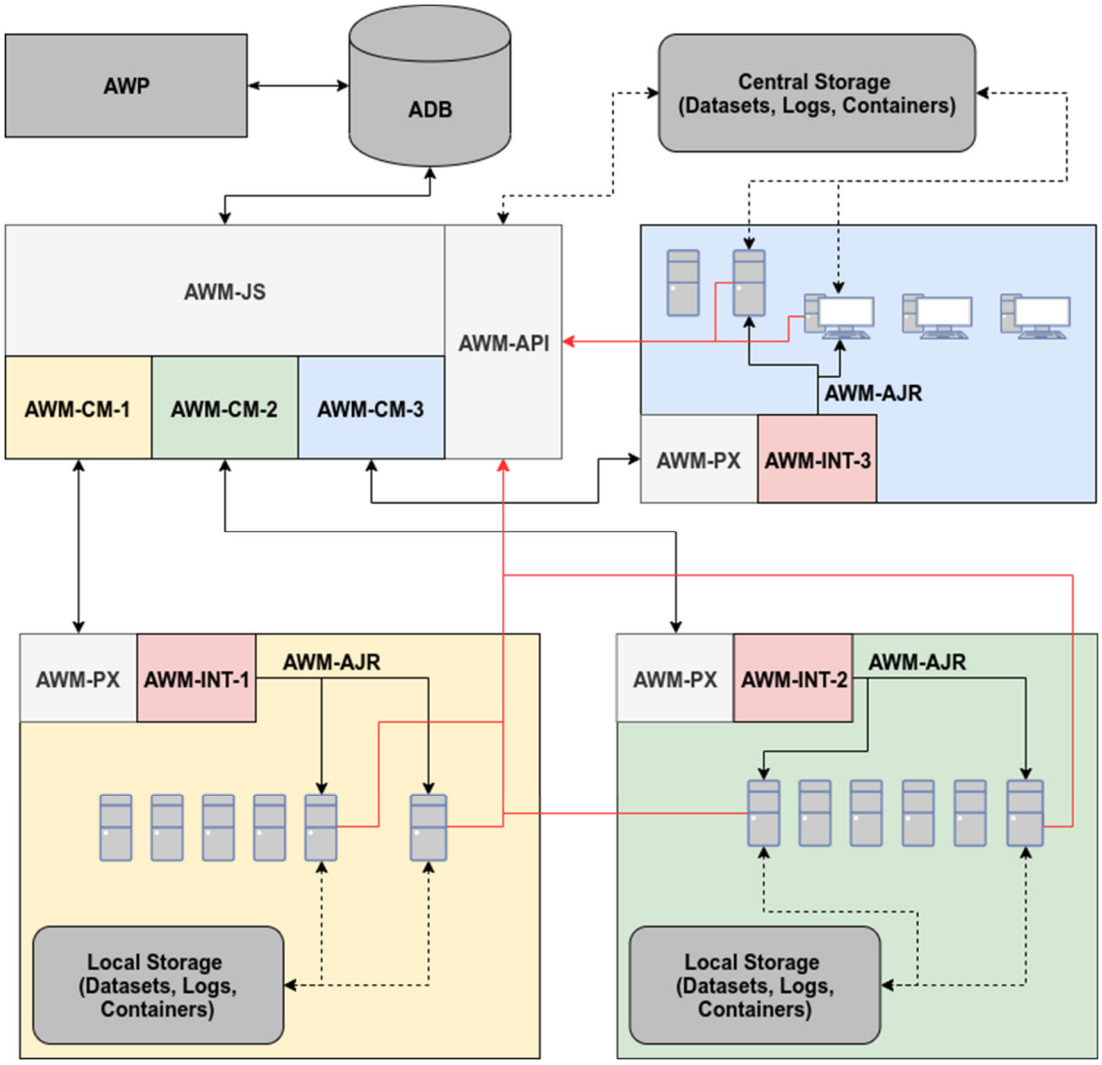

Figure 3: Axyon Platform re-shaped

Conclusions

The limitations of the original Axyon Platform made it difficult to address the growing need for computational power of AI algorithms and Big Data, needing a scalable and flexible computational architecture, with particular care to the treatment of sensitive (e.g. financial) data. Firstly, with the legacy architecture it was not possible to reliably attach remote nodes and HPC clusters to the system without dedicating a lot of time and manual work installing new connected resources with different architectures, configuring them and keeping them up-to-date. Moreover, the scheduling algorithm was decentralized, with each node connected to the system unable to retrieve information about the other resources; consequently, resources optimization was delegated to the human. The first phase of the Axyon Platform's reshape included a logical subdivision of the Platform, maintaining some of the legacy components while adding several new ones. In fact, some of the tasks of the software have been collected and reassigned among new infrastructure components, in order to logically divide the job scheduling, resource interconnection and the AI model training.

The adopted strategy is a DB-centric Master-Slave architecture, implemented in the AWM system, while keeping untouched the AWP for visual job management and the ADB for information storing. The AWM is able to obtain all the information about interconnected resources through its AWM-CM component and then to centrally manage job scheduling through the AWM-JS, optimizing resources allocation. The Slave counterpart is composed by an AWM-PX, an AWM-INT, and an AWM-JR. The first component provides a standard interface to the AWM-CM in order to establish a communication path to manage Jobs and resources, while the AWM-INT is machine-specific and is used to interact with different remote systems. The aforementioned scheme gives flexibility to easily add

any kind of resource to the Axyon Platform, thanks to the containerization of the AWM-JR, the element that takes care of running AI algorithms, while the containerization technology ensures a replicable and tested environment. The AWM-JR uses a standard API communication system to report the output and the results of the running jobs, through the component called AWM-API attached to the Master that takes care of storing the results on the ADB.

The whole infrastructure has been implemented and tested on D.A.V.I.D.E. supercomputer in CINECA, giving us the opportunity to use a standard HPC system designed for and dedicated to Artificial Intelligence. While the Master component has been hosted on an Axyon internal server, the Slave part has been deployed on D.A.V.I.D.E. supercomputer using *git*. After changing the appropriate configuration parameters in the AWM-INT and AWM-CM, we managed to reliably launch jobs on the aforementioned supercomputer. Multiple job types have been successfully submitted to D.A.V.I.D.E., after automatically checking and downloading the correct dataset and container, reporting results through the AWM-API during and after the execution.

Given the successful outcome of the project, the reshaped Platform will be adopted in production to obtain tangible business advantage by leveraging the computational power provided by external resources that will drastically reduce the training time of our AI algorithms.

References

[1] Huang, W., Lai, K.K., Nakamori, Y., Wang, S., Yu, L.: Neural networks in finance and economics forecasting. Int. J. Inf. Technol. Decis. Making 06(01), 113–140 (2007)

[2] W. A. Ahmad, A. Bartolini, F. Beneventi, L. Benini, A. Borghesi, M. Cicala, P. Forestieri, C. Gianfreda, D. Gregori, A. Libri, F. Spiga, S. Tinti, "Design of an energy aware petaflops class high performance cluster based on power architecture", *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 964-973, 2017.

[3] Singularity website: www.sylabs.io

[4] Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: Scientific containers for mobility of compute. PLoS ONE 12(5): e0177459.

[5] Azaba, A.; Muscianisi, G.; G., Wiber; , Fernandez, C., “Evaluation ofLinux Container and full virtualization for HPC Applicationsin PRACE 5IP”, 2019, PRACE Whitepaper 286, http://www.prace-ri.eu/white-papers/

[6] Yoo A.B., Jette M.A., Grondona M. (2003) SLURM: Simple Linux Utility for Resource Management. In: Feitelson D., Rudolph L., Schwiegelshohn U. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science, vol 2862. Springer, Berlin, Heidelberg

[7] Paramiko Library website: http://www.paramiko.org/

[8] NVIDIA Docker Compatibility with Singularity for HPC website, https://devblogs.nvidia.com/docker-compatibility-singularity-hpc/

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.