DEEPHEALTH

D5.4 The runtime system for DeepHealth libraries

Project ref. no.	**H2020-ICT-11-2018-2019 GA No. 825111**
Project title	Deep-Learning and HPC to Boost Biomedical Applications for Health
Duration of the project	1-01-2019 – 31-12-2021 (36 months)
WP/Task:	WP5/ T5.4
Dissemination level:	PUBLIC
Document due Date:	31/03/2020 (M15)
Actual date of delivery	06/04/2929 (M15)
Leader of this deliverable	BSC
Author(s)	Maria A. Serrano (BSC), Eduardo Quiñones (BSC)
Contributor(s)	Ignacio Penas (EPFL), Tatiana Silva (TREE)
Version	V1.0

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 82511

Document history

Version	Date	Document history/approvals
0.1	18/03/2020	First version of contents provided by BSC
0.2	23/03/2020	Private cloud API description provided by TREE
0.3	01/04/2020	Global Resource Manager description provided by EPFL
0.4	01/04/2020	Review of contents and demonstrator instructions by BSC
0.5	02/04/2020	Reviewed by UNIMORE
0.6	03/04/2020	Reviewed by TM and PC
1.0	06/04/2020	Final version ready to be submitted

DISCLAIMER

This document reflects only the author's views and the European Community is not responsible for any use that may be made of the information it contains.

Copyright

© Copyright 2019 the DEEPHEALTH Consortium

This work is licensed under the Creative Commons License “BY-NC-SA”.

Table of contents

1 Executive summary

This deliverable covers the work done in Task 5.4 "*HPC runtime support*" from month 7 to month 15. Concretely, this deliverable provides a demonstrator with a subset of the runtimes included in the DeepHealth HPC infrastructure (see Deliverable D1.2 [1] for further information), and needed to distribute the computation of ECV and EDDL libraries: The COMPSs runtime, the SLURM-based Global Resource Manager (GRM) and the Cloud infrastructure (the Mango runtime for FPGA support is described in deliverable D5.1). The demonstrator includes: (1) the enhancements introduced in COMPSs for an efficient distribution of the DeepHealth libraries across an heterogeneous computing infrastructure composed of a classical HPC and the private DeepHealth cloud infrastructure; (2) a first integration with COMPSs and the GRM; and (3) a first distributed version of the EDDLL training operation parallelised with COMPSs[1]. This deliverable also includes the instructions needed to install and run the demonstrator. A final release of the HPC runtime support will be provided at month 28 including all the functionalities expected.

The rest of this document is organized as follows: Section 2 provides a short description of the COMPSs framework. Section 3 describes the first enhancement of COMPSs to deploy applications into the different DeepHealth computing infrastructures, including the cloud infrastructure provided by the partner TREE. Section 4, describes the integration of COMPSs with the GRM developed by the partner EPFL. Section 5 provides the instructions to configure and execute the demonstrator, including a distributed version of the EDDLL training operation (see footnotes [1] and [2]). Finally, conclusion are presented in Section 6.

2 The COMPSs runtime framework

One of the main features of the COMPSs runtime framework is that it abstracts the application from the underlying distributed infrastructure; hence, COMPSs programs do not include any detail that could tie them to a particular platform, boosting portability among diverse infrastructures and enabling execution in both a classical HPC environment and a Cloud-based environment. This is a key aspect for DeepHealth.

Firstly, this section briefly describes the COMPSs programming model and the runtime included in the demonstrator and presents the EDDLL training operation parallelised with COMPS[1]. Secondly; this section presents two possible deployments of the EDDLL training operation parallelised with COMPSs and included in the demonstrator: a *classical Linux-based infrastructure* and a *private cloud-based infrastructure* provided by the partner TREE. The deployments presented in this deliverable do not use the final DeepHealth HPC infrastructure and they have been setup for demonstration purposes only.

2.1 The COMPSs framework: Task programming model and the runtime

COMPSs offers a portable programming environment based on a *task model*, whose main objective is to facilitates the parallelization of sequential source code written in Java or Python programming languages, in a distributed and heterogeneous computing environment. In COMPSs, the programmer is responsible of identifying the units of parallelism (named *COMPSs tasks*) and the synchronization data dependencies existing among them by annotating the sequential source code (using annotations in case of Java or standard decorators in case of Python).

Figure 1 shows a snipped (simplified for readability purposes) of the parallelisation of the EDDLL training operation with COMPSs. COMPSs tasks are identified with a standard Python decorator `@task` (lines 1 and 5). The `IN`, `OUT` and `INOUT` arguments define the data directionality of function parameters. By default, parameters are IN, and so there is no need to explicitly specify IN parameters.

[1] It is important to remark the EDDLL training operation is used for demonstration purposes only. Full description of the parallelisation and distribution strategy of ECVL and EDDLL will be provided in Deliverable D2.1 at month 17

Moreover, when a task is marked with `is_replicated=True`, the COMPSs task is executed in all the available computing nodes for initialization purposes; otherwise, it executes on the available computing resources. The train iterates over `num_epochs` epochs (line 14). At every epoch, `num_batches` batches are executed (line 15), each instantiating a new COMPSs task (line 16) with an EDDLL train batch operation. All COMPSs tasks are synchronize at line 18 with `compss_wait_on`, and the partial weights are collected. The gradients of the model are then updated with the partial weights at line 20.

```
1.  @task (is_replicated = True)
2.  def build (model):
3.    # The model is created at each worker
4.    […]
5.  @task(INOUT = weights)
6.  def train_batch(model, dataset):
7.    # A train operation is executed at each worker
8.    # on the model and the dataset passed
9.    […]
10. def main():
11.   # A new model is created
12.   net = eddl.model([…])
13.   build(net)
14.   for i in range(num_epochs):
15.     for j in range(num_batches):
16.       weight[j] = train_batch(net,dataset)
17.     # Synchronize all weights from workers
18.     compss_wait_on(weight)
19.     # Update weights on the model
20.     update_gradients(net,weight)
```

Figure 1. A (simplified) snipped of pyEDDLL training operation parallelised with COMPSs.

The task-based programming model of COMPSs is then supported by its runtime system, which manages several aspects of the application execution and keeps the underlying infrastructure transparent to the programmer. The COMPSs runtime is organised as a master-worker structure:

- The *master*, executed in the computing resource where the application is launched, is responsible for steering the distribution of the application and data management.
- The *worker(s)*, co-located with the Master or in remote computing resources, are in charge of responding to task execution requests coming from the Master.

One key aspect is that the master maintains the internal representation of a COMPSs application as a Direct Acyclic Graph (DAG) to express the parallelism. Each node corresponds to a COMPSs task and edges represent data dependencies (and so potential data transfers). As an example, Figure 2 presents the DAG representation of the EDDLL training operation presented in Figure 1.

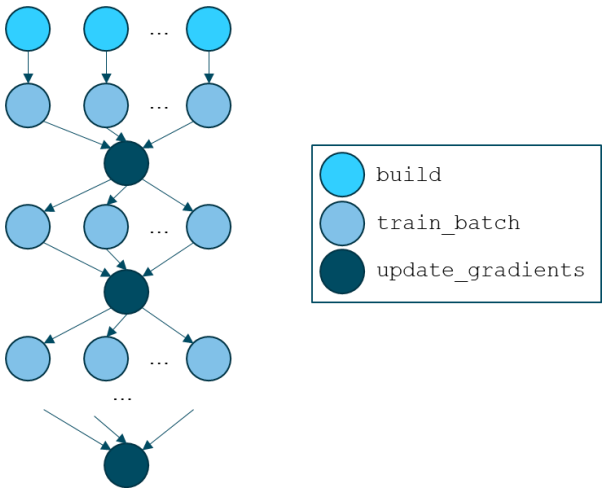

Figure 2. DAG representation of the application presented in Figure 1.

Based on this DAG, the runtime can automatically detect data dependencies between COMPSs tasks: as soon as a task becomes ready (i.e., when all its data dependencies are honoured), the master is in charge of distributing it among the available workers, transferring the input parameters before starting the execution. When the COMPSs task is completed, the result is either transferred to the worker in which the destination COMPSs tasks executes (as indicated in the DAG), or transferred to the master if a `compss_wait_on` call is invoked.

2.2 Deployment into a Classical Linux-based Infrastructure

In this scenario, the EDDLL and the ECVL will be deployed and executed natively in a Linux-like environment, a very common scenario in HPC environments. In this configuration, all the computing resources available for execution require a native installation of the COMPSs framework, and a password-less *ssh* connection is required among computing resources. Figure 3 shows an example of this deployment. The execution starts in the *Computing Resource 1*, where the COMPSs Master executes. Then four workers are deployed in four different resources to distribute the workload, where the ECVL data loading and processing and EDDLL training operations can be distributed.

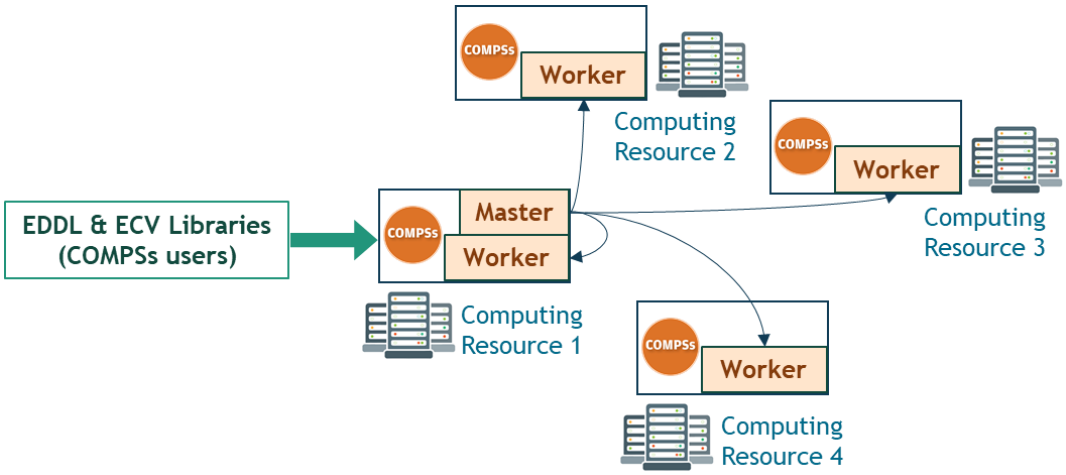

Figure 3. COMPSs deployment in a HPC infrastructure.

In other to facilitate the portability of the demonstrator without requiring the full installation of the COMPSs framework, this deployment scenario considers a Docker image of COMPSs as described below. In this case however, no container orchestrator is used as explained in the next section.

2.3 Deployment into the Private Cloud-based Infrastructure provided by TREE

The COMPSs run-time is capable of supporting containerized applications. To do so, a docker image [2] contains the needed dependencies to launch a COMPSs master and several COMPSs workers, and the final user application (in our case the EDDL and/or ECV libraries). Unlike the Linux-based infrastructure, there is no need for setting up the execution environment in all the computing resources, but only a docker image must be available, e.g., Docker Hub [3]. Figure 4Figure 3 shows an example of this deployment. The execution starts in the *computing resource 1*, where the COMPSs Master executes. Then three workers are deployed in three different containers in the cloud infrastructure (in our case the cloud is provided by TREE), where the COMPSs application is distributed (in our case, EDDLL training operations).

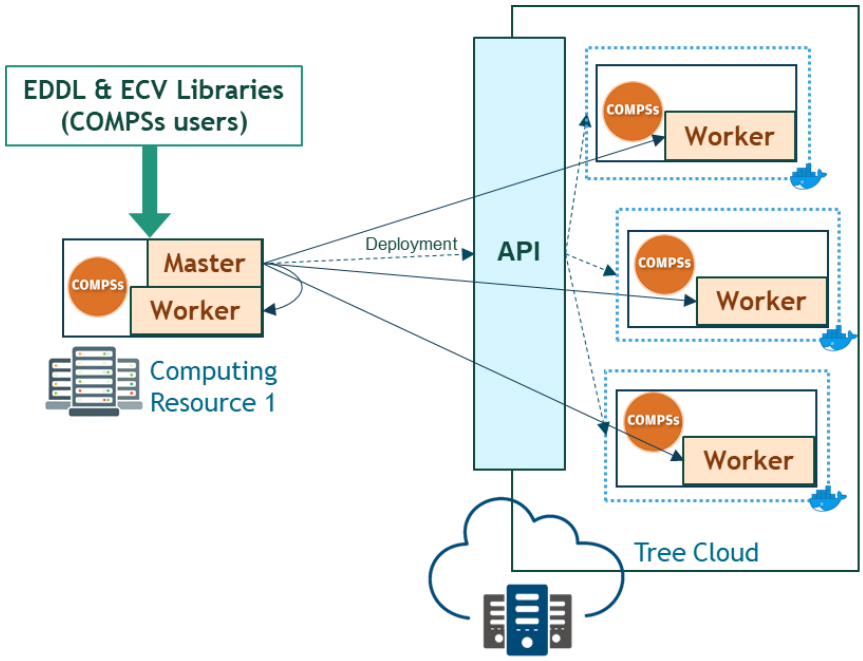

Figure 4. COMPSs deployment in a private cloud infrastructure.

The COMPSs runtime is being adapted to support the cloud infrastructure provided by TREE. The cloud is based on Kubernetes (K8S) [4], and allows to manage applications in a container technology environment and to automate the manual processes to deploy and scale containerized applications. Moreover, an API is being developed by TREE to help abstracting the user from the infrastructure itself, speeding up the processes of deployment and management of the workflows[2]. COMPSs runtime interacts with this API to deploy workers and distribute the workload.

3 Global Resource Manager (GRM)

The role of the DeepHealth GRM is the resource allocation and assignation of multiple EDDLL training operations, each parallelised and distributed with COMPSs. The GRM is based on the *SLURM* [5] open-source resource manager, by virtue of its fault-tolerant and scalable capabilities (see deliverable D1.2 [1] for a complete description). The GRM works in a master-slave manner providing: (1) a single entry- and control-point of the applications executed in the cluster and (2) a robust and centralized log report system. This section describes the GRM included in the demonstrator.

The GRM has been integrated with COMPSs using its Docker version, which allows to integrate in the same deployment command-line both resource managers when the proper cluster conditions are provided (e.g. connectivity and visibility among nodes). To do so, we have implemented a “dockerized” version of SLURM. This version permits the easy deployment of the resource manager in any cluster with an installation of docker available in every node.

With respect to the integration of the GRM with COMPSs, we have created two overlay networks, one intended for SLURM usage and one for COMPSs usage. The connection point of these networks is among SLURM Master (also called *slurmctl*) and COMPSs Master as these two containers are connected to both networks. When a new execution is requested to SLURM it assess the available resources and assigns the available ones to the requested application; then SLURM subscribes those resources to COMPSs (by writing in the file *project.xml*) and launches the COMPSs master daemon. For each new application, SLURM will call a different COMPSs master allowing, this way, the simultaneous and distributed execution of different trainings at the same time.

[2] An initial description of the Hybrid cloud computing is included in D1.2 [1]. Further details will be provided in Deliverable 5.7 (M26).

4 First release of the runtime systems for DeepHealth libraries

This section describes the instructions to deploy and execute the tree components that form the demonstrator presented in this deliverable:

1. The deployment and execution of an EDDLL training operation parallelised with COMPSs on a Linux-based and a private cloud-based infrastructures.
2. The deployment and execution of the GRM combined with COMPSs.

4.1 The COMPSs runtime on different Computing Infrastructures

This section presents how to run a preliminary version of an EDDL training operation in two different computing environments, demonstrating the heterogeneous capabilities of COMPSs. The demonstrators presented here have been executed on a Linux Ubuntu 18.04.

4.1.1 COMPSs in a Linux-based Infrastructure

1. Install Docker and docker-compose in your own computing resource. The one provided in this deliverable has been tested with Docker version 19.03.7, build 7141c199a2, Ubuntu 18.04.
2. Pull the demonstrator image by running:

   ```
   docker pull bscppc/compss-deephealth-demo
   ```

3. Download the *docker-compose.yml* file provided and store it in a separate directory named *deephealth*:

   ```
   git clone https://github.com/deephealthproject/docker-compss-runtime
   ```

4. Run:

   ```
   docker-compose up -d –scale compss-worker=4
   ```

   This will deploy five containers, four of which will take on the role of COMPSs workers, while the remaining will be used as the COMPSs master.

5. Access the COMPSs master container by running:

   ```
   docker exec -it deephealth_compss-master_1 bash
   ```

   This will open a bash session inside the container, in the directory with the EDDL application and the needed COMPSs configuration options.

6. Enter the pyeddl directory:

   ```
   cd pyeddl/third_party/compss_runtime
   ```

7. Launch COMPSs by running the following command:

   ```
   runcompss --lang=python --python_interpreter=python3 –project=linux-based/project.xml --resources=linux-based/resources.xml eddl_train_batch_compss.py
   ```

8. Once the application finishes correctly, the COMPSs master container can be exited. The containers can be destroyed by running, in the same directory:

   ```
   docker-compose down -v
   ```

4.1.2 COMPSS in a cloud

Follow steps 1 and 2 of previous Section 4.1.1, and then:

1. Make sure that *kubectl*, *openvpn* and *curl* package are installed in your Linux environment: *curl* is generally present in all Linux distributions; *kubectl* can be installed through the universal package

manager *snap*, and *openvpn* can be obtained through the default package manager of the operating system, such as *apt*.

2. Download the files required for the VPN connection into a separate folder (see in the comments attached to this deliverable how to get these files and the credentials; this information has not been included here for security reasons). In a terminal, run the following command, to connect the VPN:

   ```
   sudo openvpn --script-security 2 --config ./infierno-TCP-1199-deephealth.ovpn
   ```

3. Download the configuration files by executing the following command:

   ```
   git clone https://github.com/deephealthproject/docker-compss-runtime/tree-cloud
   ```

4. A container must be deployed in the cluster, which will play the role of the COMPSs master. This can be done by running the provided script:

   ```
   deploy_master.sh
   ```

   which communicates with the API developed by TREE.

5. Disconnect from the VPN.

6. Copy the provided *kubeconfig* configuration file in a separate, empty folder. Use *kubectl*, along with the aforementioned configuration file, to access the newly created COMPSs master, by running the following command:

   ```
   KUBECONFIG=kubeconfig kubectl exec -it compss-master – /bin/bash
   ```

7. Inside the COMPSs master container, go to *pyeddl/third_party/compss_runtime* directory and launch the execution by running the following command*:*

   ```
   runcompss --lang=python --python_interpreter=python3 --project=cloud/project.xml --resources=cloud/resources.xml eddl_train_batch_compss.py
   ```

4.2 The Global Resource Manager and COMPSs

This section presents the instructions to deploy the demonstrator that executes a sample application using DeepHealth GRM. In this case, the EDDLL training operation is not used.

4.2.1 Software requirements

1. Install Docker on all the servers in which the demonstrator will be deployed. The one provided in this deliverable has been tested with Docker version 19.03.7, build 7141c199a2, CentOS-release-7-7.1908.
2. Download docker repos for GRM and COMPSs in all servers:
   a. COMPSs master: `docker pull bscppc/compss-container-master`
   b. COMPSs worker (sample): `docker pull bscppc/compss-worker-matmul`
3. SLURM GRM (All in one):
   a. Get source code executing the following command:
      ```
      git clone https://github.com/deephealthproject/docker-compss-runtime.git
      ```
   b. Go to the *docker-compss-runtime/slurm* directory.
   c. Run:
      ```
      docker build -t {tag_name}
      ```

d. Distribute the image among the servers. The name of the image should correspond with the name assigned on the deployment file in image field.

4.2.2 Docker swarm environment creation

4. Grant the corresponding visibility among nodes opening the required ports[3] for docker swarm.
5. Create a docker swarm network in which the advertised address of the master node must be visible to the other nodes

   ```
   docker swarm init --advertise-addr [IP]
   ```

   (Where [IP] is the corresponding IP address of the Master)
6. Join the other nodes to the swarm using the command provided with the key in the previous step:

   ```
   docker swarm join --token {Token} {ip_address:2377}
   ```
7. Check all nodes are properly connected to the swarm running the following command in the master node:

   ```
   docker node ls
   ```

   All nodes should show as "*Ready*".

4.2.3 Create your own deployment file

8. We provide the file "*./deployment_files/vm-4-node-config.yml*" as example. In this case, the node *centos0* is the master node and the rest nodes (*centos[1-3]*) are workers. In general, only the host name should be modified to deploy the system.

4.2.4 Overlay network configuration

9. This project uses two overlay networks: one for SLURM and one for COMPSs. To create them, run to following commands in the master node:

   ```
   docker network create -d overlay --attachable --subnet=20.2.0.0/16 compss_overlay
   docker network create -d overlay --attachable --subnet=20.1.0.0/16 slurm_overlay
   ```

4.2.5 Deploy the cluster into the swarm

10. In the master node execute the following command:

    ```
    docker stack deploy -c {path_to_config_file} {stack_name}
    ```
11. Check the correct service deployment with:

    ```
    docker service ls
    ```

    All the services must have 1/1 replicas.

4.2.6 Configure COMPSs

12. Copy id_rsa.pub from *compss_master* docker container to every node in order to permit passwordless ssh connection.

4.2.7 Run sample application inside the *slurmctld* docker container

13. Execute the following command:

    ```
    cd /root/
    sbatch slurm-compss.sh
    ```

[3] See https://docs.docker.com/engine/swarm/swarm-tutorial/#open-protocols-and-ports-between-the-hosts

5 Conclusions

This deliverable has presented the advances done in Task 5.4 "*HPC runtime support*" (from month 7 to month 15) on the development of a HPC runtime framework for the deployment, distribution and execution of the ECV and EDDL libraries. Concretely, this deliverable has briefly described the enhancements on the COMPSs runtime done by BSC, the SLURM-based GRM done by EPFL and the Cloud infrastructure done by TREE. Moreover, this deliverable considers a preliminarly parallel implementation of the EDDLL training operation for demonstration purposes only.

The complete DeepHealth HPC runtime framework will be delivered at M28 on the different HPC infrastructures provided by the DeepHealth partners (see Deliverable D1.2 [1] for a complete list of HPC insfrastructure). A complete description of the parallelisation and distribution strategy of the ECV and the EDDL libraries will be provided in Deliverable D2.1 at month 17.

6 Bibliography

[1] E. Quiñones, "D1.2 HPC infrastructure and application adaptation requirements," DeepHealth project, June 2019.

[2] "Docker," [Online]. Available: https://www.docker.com/.

[3] "Docker Hub," [Online]. Available: https://www.docker.com/products/docker-hub.

[4] "Kubernetes (K8s)," [Online]. Available: https://kubernetes.io/.

[5] "SLURM," [Online]. Available: https://slurm.schedmd.com/overview.html.

[6] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes and J. Labarta, "Pycompss: Parallel computational workflows in Python," in *The International Journal of High Performance Computing Applications*, 2017.

[7] "COMPSs documentation, version 2.6," BSC, [Online]. Available: https://compss-doc.readthedocs.io/en/2.6/.