Automatizing the creation of specialized HPC containers

International Journal of High Performance Computing Applications
XX(X):1–12

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Jorge Ejarque[1] and Rosa M Badia[1]

Abstract

With Exascale computing already here, supercomputers are systems every time larger, more complex, and heterogeneous. While expert system administrators can install and deploy applications in the systems correctly, this is something that general users can not usually do. The eFlows4HPC project aims to provide methodologies and tools to enable the use and reuse of application workflows. One of the aspects that the project focuses on is simplifying the application deployment in large and complex systems. The approach uses containers, not generic ones, but containers tailored for each target High-Performance Computing (HPC) system. This paper presents the Container Image Creation service developed in the framework of the project and experimentation based on project applications.

We compare the performance of the specialized containers against generic containers and against a native installation. The results show that in almost all cases, the specialized containers outperform the generic ones (up to 2× faster), and in all cases, the performance is the same as with the native installation.

Keywords

HPC applications and workflows, Architecture-specific containers, Automatic deployment, Exascale systems, Heterogeneous workflows

Introduction

One of the barriers to adopting High-Performance Computing (HPC) is the complexity of developing, deploying and executing complex workflows in federated HPC environments. New scientific and industrial applications require implementing workflows that combine traditional HPC simulation and modelling with big data analytics (DA) and machine learning (ML) algorithms. Integrating these different technologies in a single workflow increases the complexity of managing its entire lifecycle. Starting from the development phase, integrating different HPC, DA and ML phases requires additional programming efforts, for example, by introducing additional glue code, which deals with the execution and data integration between the different parts of the workflow. In the deployment phase, users need to perform complex software installations in HPC systems beyond their technical skills. Nowadays, this is performed in most cases as a manual process by system administrators. Having the workflows ready for execution in a supercomputer can take time and human resources, which increases if there is a need for replication in several clusters, for example, due to reliability requirements. Finally, in the execution phase, all the different components must be dynamically and intelligently orchestrated to use resources efficiently.

The eFlows4HPC project Ejarque et al. (2022) aims to widen the access to HPC to newcomers and, in general, simplify the development, deployment and execution of complex workflows in HPC systems. It proposes to simplify this process in two ways. From one side, the eFlows4HPC software stack aims to provide the required functionalities to manage this complex workflow's lifecycle. On the other side, it introduces the HPC Workflow as a Service (HPCWaaS) concept, which leverages the software stack to widen access to HPC by the different communities. This service offering tries to bring the Function as a Service (FaaS) concept to the HPC environments trying to hide all the complexity of an HPC Workflow execution to end users. Through three application Pillars with high industrial and social relevance (manufacturing, climate and urgent computing for natural hazards), the project demonstrates how the implementation of forthcoming efficient HPC and data-centric applications can be developed with the proposed novel workflow technologies.

The article focuses on the deployment phase of the eFlows4HPC methodologies, which are based on the use of container images and their automatic creation. Standard creation processes for container images are normally relying on OS packages which are compiled for generic architectures in order to maximize the package portability. However, these packages are benefiting of particular features of HPC processors, such as vector instructions. Moreover, OS distributions only provide packages for certain library versions. If you want to use other versions, the container image creation must be compiled from sources. In this case, the compilation flags must be carefully selected. Otherwise, images would be created that would not be generic enough and could only be used on the same architecture as the

[1]Barcelona Supercomputing Center, Spain

Corresponding author:
Jorge Ejarque, Barcelona Supercomputing Center, Workflows and Distributed Computing, Barcelona, 08034 - Spain
Email: jorge.ejarque@bsc.es

machine that is building the container image. This library version limitation can also affect the usage of specific hardware available in HPC clusters, such as network fabric or accelerators. To benefit from this hardware, the versions of some libraries (MPI, CUDA, etc) must be compatible with the ones installed in the HPC Cluster.

In this article, we present a Container Image Creation service that takes into account all these issues. It leverages HPC and multi-platform container builders to automate the creation of container images tailored to specific HPC platforms. The contributions of this article are:

- Design and implementation of a Container Image Creation (CIC) service for HPC systems
- Integration of the CIC service in the eFlows4HPC deployment process
- Validation through real use cases from manufacturing, climate modelling and urgent computing.

The structure of the paper is the following: the next section presents state-of-the-art, and some background description about the eFlows4HPC project follows it. Next, the Container Image Creation service, which is the article's core, is described. We also present some experimentation with application workflows from the project, and some final remarks conclude the paper.

State of the Art

HPC builder systems

Spack Gamblin et al. (2015) is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments. It was designed for large supercomputing centres, where many users and application teams share common installations of software on clusters with exotic architectures, using libraries that do not have a standard Application Binary Interface (ABI). Spack is non-destructive: installing a new version does not break existing installations, so many configurations can coexist on the same system.

Most importantly, Spack is simple. It offers a simple spec syntax so that users can specify versions and configuration options concisely. Spack is also simple for package authors: package files are written in pure Python, and specs allow package authors to maintain a single file for many different builds of the same package.

More specifically, for container image creation, Spack provides the Spack Environments. An environment is used to group together a set of specs for the purpose of building, rebuilding and deploying in a coherent fashion. Environments separate the steps of (a) choosing what to install, (b) concretizing, and (c) installing. This allows Environments to remain stable and repeatable. Also, environments allow several specs to be built at once. In addition, an Environment that is built as a whole can be loaded as a whole into the user environment. The environment to be created is defined in a manifest file in YAML format (spack.yaml).

EasyBuild Geimer et al. (2014) is a software build and installation framework that allows you to manage (scientific) software on High-Performance Computing (HPC) systems in an efficient way. Easybuild provides a flexible framework for building and installing (scientific) software, that fully automates software builds, diverting from the standard configure, make, make install with custom procedures. Easybuild allows for easily reproducing previous builds, keeps the software build recipes and specifications simple and human-readable, supports the co-existence of versions and builds via dedicated installation prefix and module files and enables sharing with the HPC community.

HPC Container Maker (HPCCM) McMillan (2018) is an NVIDIA open-source tool to make it easier to generate HPC application container specification files. It is able to generate either Dockerfiles or Singularity definition files from the same Python recipe. It is based on the idea of having available a library of HPC building blocks that are used in the Python recipe. While the building blocks have a default configuration, they are also fully configurable with multiple options. The set of building blocks includes, for example, multiple MPI versions, compilers, HPC libraries such as MKL or OpenBLAS, etc. CUDA is always included through the base image used to build the image.

Since the methodology described in this paper is agnostic of the builder system, we can use any of the former methodologies. However, so far, we have used Spack. We have found that Spack is more generic when defining the `spack.yaml` and requires one single `package.py` per software component. HPCCM mostly focuses on GPU-based target systems. Easybuild requires specifying each *variant* as a separate package. Another drawback of Easybuild is that it requires the user to define the compiler flags for each architecture, while Spack can determine this automatically. In an automated environment like the one we have designed, it was easier to base the implementation on Spack.

Container builder systems

Docker Build Merkel (2014) is one of Docker Engine's most used features. When creating an image, Docker Build is used. Build is a key part of the software development life cycle that allows packaging and bundling code and shipping it anywhere. The most common method of executing a build is by issuing a docker build command. The Command Line Interface (CLI) sends the request to Docker Engine which, in turn, executes the build. Recently Moby BuildKit has been added to the Docker Engine, a new component to build images.

The new client Docker Buildx, is a CLI plugin that extends the docker command with the full support of the features provided by the BuildKit builder toolkit. `docker buildx build command` provides the same user experience as `docker build` with many new features, like creating scoped builder instances, building against multiple nodes concurrently, outputs configuration, inline build caching, and specifying target platform. In addition, Buildx also supports new features that are not yet available for regular docker build like building manifest lists, distributed caching, and exporting build results to Open Container Initiative (OCI)* image tarballs.

The `build` command from Singularity Kurtzer et al. (2017) is able to download and assemble existing containers

*OCI website: https://opencontainers.org

from external resources like the Container Library and Docker Hub, and it can also be used to convert containers between the formats supported by Singularity. In addition, it can be used in conjunction with a Singularity definition file to create a container from scratch and customize it. However, the feature that enables to perform remote builds using build hosts of different architectures is only available in the enterprise license.

Buildah[†] is a tool that facilitates building OCI container images. Buildah is an open-source, Linux-based tool that can build Docker and Kubernetes-compatible images and is easy to incorporate into scripts and build pipelines. In addition, Buildah has overlap functionality with other container-related tools such as Podman[‡], Skopeo[§], and CRI-O[¶]. Buildah can create a working container from scratch but also from a pre-existing Dockerfile. In addition, it does not need a daemon running in the system when building container images. Buildah also supports building and pushing multi-architecture container images, but it is less streamlined compared to docker buildx.

Container deployment environments

The use of containers is common in the area of HPC. For example, in Olaya et al. (2022) the authors present an environment based on fine-grained containerization of both data and applications which automatically creates data lineage and record trail of workflow executions, enabling traceability of data and explainability of results. However, very few approaches to automating its deployment can be found in the literature. EASEY introduces a framework to enable container applications based on Docker to be transformed automatically to Charliecloud containers and executed on HPC systems via an integrated management system Höb and Kranzlmüller (2020). The architecture of the EASEY system is integrated as two building bricks in the layered HPC architecture. On the Applications and Users layer the EASEY-client is mainly responsible for a functional build based on a Dockerfile and all information given by the user. The middleware on the local resource management layer takes care of the execution environment preparation, the data placement and the deployment to the local scheduler.

Background: The eFlows4HPC project

Architecture

Figure 1 shows the overview of the eFlows4HPC software stack. It includes a set of software components organised in different layers: The first layer provides the syntax and programming models to implement and automatically operate complex workflows combining typical HPC simulations with HPDA and ML. The second layer consists of a set of services, repositories, catalogues, and registries to facilitate the accessibility and re-usability of the implemented workflows, software components, data sources and results. Finally, the lowest layers provide the functionalities to automate the deploy and execution of the workflow. This layer provides the components to orchestrate the deployment and coordinated execution of the workflow components in federated computing infrastructures. Moreover, it provides a set of components to manage and simplify the integration of large

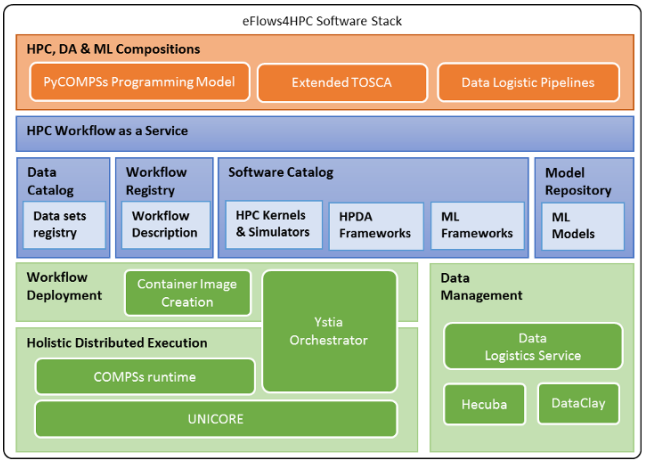

Figure 1. The eFlows4HPC software stack architecture

volumes of data from different sources and locations with the workflow execution. Actions in the layer are executed according to the workflow description provided in the first layer and to the metadata stored in the second layer services.

A workflow in eFlows4HPC is composed of three different parts: the computational workflow, developed in PyCOMPSs Tejedor and et al. (2017); the data logistic pipelines, that describe the necessary data transfers between external servers and the HPC system; and the topology of the overall workflow, described in TOSCA OASIS (2022) (see the top layer of the software stack in Figure 1). Once developed, a workflow is stored in the Workflow registry to enable its deployment and later execution.

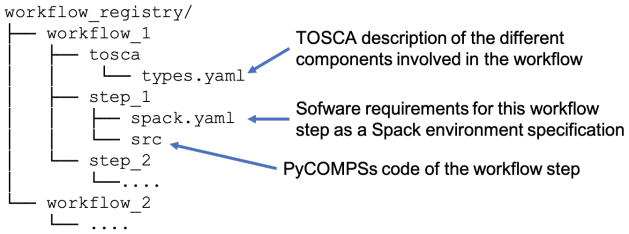

Figure 2. Workflow registry structure. Workflows are stored in a Git repository with the structure shown in the figure

The Workflow registry is a Git repository which allows developers to store the developed workflows. Workflows are stored in different folders following the structure indicated in Figure 2. Inside the workflow folder, there is a `tosca` subfolder which stores the TOSCA topology of the workflow, including references to the data logistic pipelines and the PyCOMPSs computational workflows as steps of this global workflow. Then, there is a set of folders that include the PyCOMPSs workflow source code and the software requirements of each workflow step. The data logistic pipelines are registered in the Data Logistic Service (DLS). The TOSCA workflow includes the corresponding URL of the DLS and the identifier of the pipeline to be executed.

[†]Buildah website: https://buildah.io/
[‡]Podman website: https://podman.io
[§]Skopeo website:https://github.com/containers/skopeo
[¶]CRI-O website:https://github.com/cri-o/cri-o

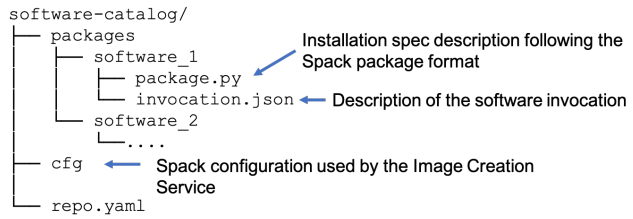

Figure 3. Software catalog structure. The workflows' software components are stored in a GIT repository with the structure shown in the picture

To include new workflows in the registry, workflow developers have to create a new fork or branch of the Git repository. In this fork/branch, they have to include a new folder for the workflow with a subfolder for the TOSCA description and the different workflow steps, as explained above. Finally, to make it available for the community, they have to create a pull request of the branch/fork containing the new workflow description to the main branch. This pull request will be reviewed by the community and included in the repository.

The Software Catalogue is a Git repository that allows software owners and workflow developers to store the description of the software used in the workflows in a way that the eFlows4HPC Software Stack can manage it transparently to final users. Figure 3 shows an example of software descriptions. It follows a structure compatible with software repositories of HPC builder systems such as Spack or Easybuild. In the case of the figure, we show an example for the Spack system. All software packages are stored in a `packages` folder. In each folder, there is a subfolder per software package. In each subfolder, developers have to provide: the `package.py` file, which includes the installation description according to the Spack schemas, and different `invocation.json` files to describe the different ways to invoke the software.

To include new software packages in the catalogue, developers have to create a new fork or branch of the git repository. In this fork/branch, they have to include a new subfolder for the software with the installation and invocation descriptions, as explained above. Finally, to make it available for the community, they have to create a pull request of the branch/fork with the new software description to the main branch. This pull request will be reviewed by the community and included in the repository.

HPC Workflows as a Service

The HPC Workflow as a Service (HPCWaaS) provides the interfaces for workflow developers and final users to manage the different steps of the workflow's lifecycle. It comprises two subcomponents that offer the interfaces according to the user role. Workflow developers interact with Alien4Cloud[‖] to develop and deploy workflows. Final workflow users interact with the Execution API to execute the deployed workflows.

Alien4Cloud is a web-based GUI which allows developers to create and deploy workflows as TOSCA topologies in a user-friendly way. For a given workflow, the TOSCA topology describes how to perform its deployment and operation procedures in different infrastructures (HPC sites). At development, it allows developers to create topologies from scratch and save them in the Workflow Registry or reuse existing ones to create new workflows. At installation time, it enables the deployment of the same workflow in different (new) environments.

The Execution API is a REST API and a CLI, which allow the workflow users to check the deployed services in the different environments, manage the credentials and execute the deployed workflows using these credentials.

Deploying workflows with eFlows4HPC

Figure 4 provides an overview of how components interact to provide the deployment functionalities. When developers want to execute a workflow, they use the Alien4Cloud interface of the HPCWaaS to indicate the workflow to deploy, select the environment (computing infrastructure) to deploy it and provide their access token. As a result of this interaction, the Alien4Cloud will retrieve the TOSCA description of the workflow (Step 1) and contact the Ystia Orchestrator (Yorc). Yorc is in charge of orchestrating the deployment of the main workflow components in the computing infrastructures and managing their lifecycle following the TOSCA part description (Step 2). The actions orchestrated by Yorc include the interactions with the Container Image Creation component to create the container images for the selected environment (Step 3). It also includes the interactions with the Data Logistics Service to set up the data pipelines that transfer the generated container images and other datasets or models that are required to execute and specific workflow (Step 4). The access to the HPC infrastructure can be either through the SSH/SCP protocols, which is the typical access protocol to these types of infrastructures, or through the Unicore services in case this middleware is available in the HPC infrastructure.

The Container Image Creation Service

The Container Image Creation (CIC) service is a component that automates the creation of container images tailored to a specific HPC platform (HPC ready containers). This component leverages specialised HPC builders (such as Spack or Easybuild), and multi-platform container build tools (such as buildx). It also uses as input the information provided by the eFlows4HPC Software Catalogue and Workflow Registry to automatically create optimised container images required to execute a workflow in a specific HPC machine.

Given a workflow registered in the Workflow registry (see fig 5) and a description of a target platform (such as CPU architecture, available MPI versions and accelerators), the CIC orchestrates the creation of the workflow container images for this platform. Based on a generated container building environment and a recipe with the required software, executes this recipe in the multi-platform container build tool. At the end of this process, the generated image will be stored together with metadata description to avoid creating the same image again for a similar platform.

[‖] Alien4Cloud website, https://alien4cloud.github.io

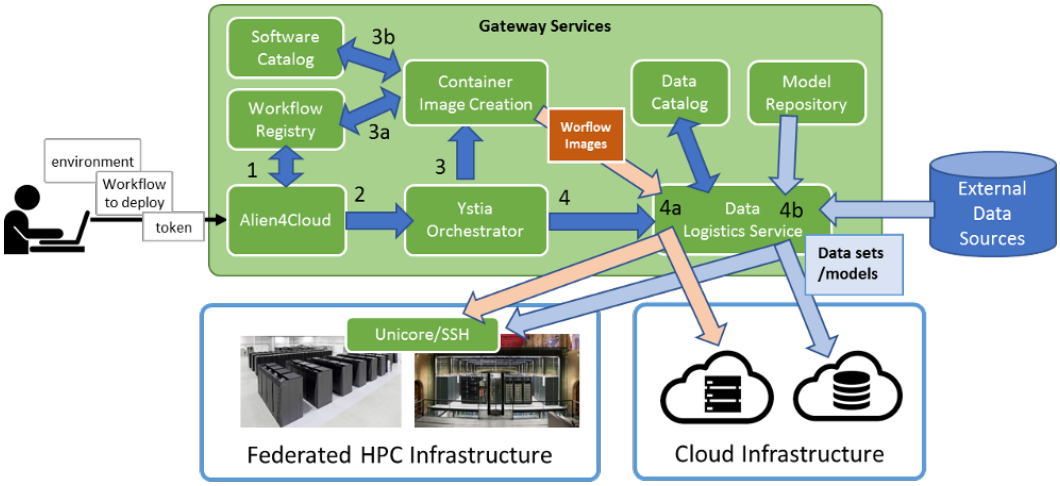

Figure 4. Deployment overview. The figure describes how developers interact with Alien4Cloud to deploy workflows and how the gateway services interact between them once a request is received.

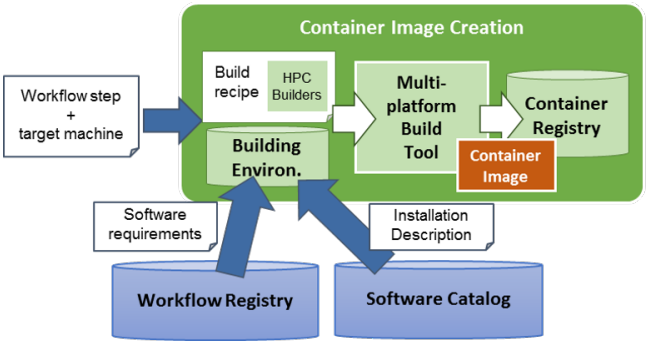

Figure 5. Container Image Creation service overview. The service has as input a workflow step and a target system. It is able to generate HPC tailored container images by using a multi-platform build tool (buildx) and HPC builder (Spack).

The service is implemented as a Flask service that enables the request to build a container image, check the status of a container building and download the image. As multi-platform container builder, we have selected Docker Buildx, and, as HPC builder, we have chosen Spack.

HPC Ready Container Image Creation Process

To build the container image, the service receives a JSON document (see Figure 6) describing the requested characteristics: the target machine description, workflow and workflow step. The target machine description indicates different properties required by the CIC to create the container. This description includes the supported container engine (Docker and Singularity are currently supported), the specific processor architecture** and the available MPI versions and runtime libraries for accessing the accelerator (e.g NVIDIA CUDA or AMD ROCm) for which the container is needed. The workflow and workflow step identify the specific step in a particular workflow that the container image corresponds to.

With this information, the CIC service obtains from the Workflow registry the `spack.yaml` file for the specific workflow step. This file is a Spack environment manifest that lists the software required to build the container image for this workflow step. The service also obtains from the Software Catalogue the `package.py` for the software packages used in the step. This Python file is written following the Spack spec syntax and describes how to build the package. It allows specifying how to build different software versions (i.e., version 1.1, version 1.2.3) or different variants. The possibility of defining different variants is very relevant for HPC since they allow specifying that a version is using MPI or CUDA (including specific versions for these libraries).

The JSON file provided to the CIC service also gives information about the target platform, architecture and container engine for which the container image is needed. The service uses this information to particularize the `spack.yaml` file for the specific target system.

The JSON file also specifies if the software requires MPI or CUDA libraries since it is necessary to build the container image with the specific library available in the host. In this case, the JSON may include which specific MPI and CUDA library is needed and the service will propagate this information to the Spack environment, compiling for the specific variant of the required software. An example of how the original Spack environment is particularized for a target HPC machine is depicted in Figure 7.

Once the Spack environment is particularized, the CIC service creates a container creation context for the multi-platform image creation with Docker Buildx. This context includes the workflow step directory with the particularized Spack environment, the package descriptions included in the Software Catalog and a Dockerfile with the Spack environment installation instructions as the one shown in Figure 8.

** Available architectures: https://spack.readthedocs.io/en/latest/basic_usage.html#support-for-microarchitectures

```
{
    "machine": {
        "platform": "linux/<amd64|386|arm[64|v6|v7]|riscv64|ppc64le|s390x>",
        "architecture": "<Spack supported archictures>",
        "container_engine": "<docker|singularity>",
        "mpi": "<openmpi|intel-mpi|mpich>@<version>",
        "gpu": "<cuda|rocm>@<version>"
        },
    "workflow": "<Folder in Workflow Registry>",
    "step_id": "<Step the workflow folder>"
}
```

Figure 6. Container configuration (JSON file. It specifies the target system's architecture and specific libraries needed (for MPI or GPU) and identifies the actual workflow step software to be built in the container image.

```
spack:
    specs:
        - fesom2
```

(a) Original Spack Environment

```
spack:
    concretizer:
        unify: true
    packages:
        all:
          target:
              - skylake
    specs:
        - fesom2
        - intel-mpi@2018.4
```

(b) Generated Spack Environment

Figure 7. Spack Environment Particularization for specific HPC machine

```
FROM ghcr.io/eflows4hpc/spack_base:0.18.1

COPY %WF_STEP% /%WF_STEP%
COPY software-catalog /software-catalog
COPY .spack /.spack

RUN  spack -C /.spack find &&  \
    && spack -C /.spack -e /%WF_STEP%
    ↪  install --fail-fast -j2 \
    && spack clean && spack gc -y

RUN  echo ".
↪  /opt/spack/share/spack/setup-env.sh &&
↪  cd /%APPDIR% && spack -C /.spack env
↪  activate ." >>
↪  /etc/profile.d/%WF_STEP%_env.sh
```

Figure 8. Dockerfile with Spack environment installation

Then, the service invokes the Docker Buildx from the container context indicating the platform required for the target machine as indicated in Figure 9. After this execution the image ready for a Docker engine is obtained. If the machine supports other container engines, such as Singularity, it can be converted to other formats as in the case of the figure.

Once the container has been built it will be deployed in the HPC system and pushed into an image repository, with the goal of retrieving it in case it is needed in the future.

Sample CIC service client

In order to simplify the access to the CIC service, we have developed a simple `bash` client. Figure 10 shows the usage

```
cd build_context_f23pdx45t
docker buildx build --progress plain
↪  --builder multi-platform --platform
↪  linux/amd64 --load --rm -t esm_skylake
↪  -f Dockerfile .
if [ "$engine" == "singularity" ]; then
    singularity build
    ↪  $IMAGES_PATH/esm_skylake.sif
    ↪  docker-daemon://esm_skylake:latest
fi
```

Figure 9. Sample buildx Command invoked from the CIC service

of the client. It receives as input the service URL, and the action to execute (build image, check status or download the image). Depending on the actual action, it receives also the path of the JSON file, the identifier of the image or the path where to download the image. Examples of the different commands are shown in Figure 11.

```
cic_client.sh <image_creation_service_url>
↪  <"build"|"status"|"download">
↪  <json_file|build_id|image_name>
```

Figure 10. Client usage for the CIC service

Experiments

Experimental setting

To validate the features of the Container Image creation system, we have selected a set of use cases and

```
$> cic_client.sh https://bscgrid20.bsc.es build exageostat_request.json
Response:
{"id":"f1f4699b-9048-4ecc-aff3-1c689b855adc"}

$> cic_client.sh https://bscgrid20.bsc.es status f1f4699b-9048-4ecc-aff3-1c689b855adc
Response:
{"filename":"reduce_order_model_sandybridge.sif",
↪  "image_id":"ghcr.io/eflows4hpc/reduce_order_model_sandybridge",
↪  "message":null,"status":"FINISHED"}

$> cic_client.sh https://bscgrid20.bsc.es download exageostat_skylake.sif
Response:
Resolving bscgrid20.bsc.es (bscgrid20.bsc.es)... 84.88.52.251
Connecting to bscgrid20.bsc.es (bscgrid20.bsc.es)|84.88.52.251|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2339000320 (2.2G) [application/octet-stream]
Saving to: 'exageostat_skylake.sif'

exageostat_skylake.sif        0%[                              ]   4.35M   550KB/s    eta 79m
↪  0s
```

Figure 11. CIC client sample commands

heterogeneous systems. We have generated two container images for each case and system, one using our approach and another with a standard container build process with generic packages. We compare the execution with these container images against the native installation in the respective HPC clusters.

The applications evaluated are the use cases of the eFlows4HPC project. These applications use a wide variety of HPC features suitable for evaluating the system. The first application (Digital twins) mainly uses shared memory parallelism with OpenMP and Linear Algebra systems solved using the Eigen and BLAS libraries. The main kernel of the second application (Earth System Model) mainly uses distributed memory parallelism with MPI and BLAS libraries. Finally, the main kernel of the third application (PTF/FTRT) uses GPUs to perform its computations.

Regarding the computing infrastructures, we have used: the General purpose cluster of MareNostrum 4, the CTE-Power Cluster and the Nord3 supercomputer. All of them are hosted by the Barcelona Supercomputing Center and support the Singularity container engine. The General Purpose Cluster MareNostrum 4 (MN4)[††] is composed of 3456 compute nodes with two 24-core Intel Xeon Platinum 8160 CPU (Skylake) each, with 96 GB of main memory and 100 Gbit/s Intel Omni-Path interconnect. The CTE-Power Cluster[‡‡] is composed of 52 computing nodes, each with two 20-core IBM Power9 8335-GTH20 512GB and 4 GPU NVIDIA V100 (Volta) with 16GB HBM2. Finally, the Nord 3 cluster comprises 84 IBM dx360 M4 compute nodes with two Intel 8 core SandyBridge EP processors each, with 32GB connected through a 100Gbit/s Infiniband Network.

Digital twins in manufacturing: eFlows4HPC Pillar I application

The Pillar I of the eFlows4HPC project aims to develop a flexible workflow for the construction of Reduced Order Models (ROM) to define Digital Twins for manufacturing applications Boschert et al. (2021).

The process starts by defining a detailed model: the Full Order Model (FOM). The FOM considers multiple scenarios, storing the results in a database. The results are collected to find common patterns in the solution, defining the ”reduced basis” used to construct the ROM. The ROM supports approximate predictions at a much reduced computational cost. Refining the model leads to a hyper-reduced model to further reduce the computational cost. The hyper-reduced model results are compared to the FOM ones to decide if further iterations are needed or if the model is final.

The FOM is built through a training campaign, which defines a set of simulations of the KRATOS Multiphysics software Dadvand et al. (2013, 2010). These results all together form a large data matrix used to generate the ROM. For the example shown in this article, the ROM is obtained by implementing a Randomized Singular Value Decomposition (RSVD) method. The RSVD has been implemented as a Python script that combines PyCOMPSs tasks and dislib methods. The dislib Álvarez Cid-Fuentes et al. (2019) is a distributed machine learning library implemented on top of PyCOMPSs. The next step generates new auxiliary data using the computed ROM, used to build the hyper-reduced model. The hyper-reduced model is validated by assessing its performance against the FOM data. According to the results of this step, the workflow may need to go back to the beginning and improve the initial training set data. Once the model is final, it can be deployed in the cloud or in edge devices close to the manufacturing environment. This generic workflow is applied in the framework of the project to model the overheating of industrial electrical engines. For the safe

[††] MareNostrum 4 system overview:
https://bsc.es/supportkc/docs/MareNostrum4/intro
[‡‡] CTE-Power system overview:
https://bsc.es/supportkc/docs/CTE-POWER/intro
Nord3 system overview:
https://bsc.es/supportkc/docs/Nord3v2/intro

operation of the type of engine considered in the project, the temperature in the windings must not exceed a critical temperature, as the electrical insulation will be damaged due to thermal degeneration.

Image Creation and Deployment A prototype version of the workflow has been developed and it is available in the eFlows4HPC Workflows Repository. We have tested the CIC service with this workflow. To invoke the service, a specific JSON file is provided to the service (see figure 13). The information in the JSON file is used to retrieve from the workflow repository the `spack.yml` file, which is shown in figure 12. This file lists the three different software components that are used in the workflow: PyCOMPSs, dislib and Kratos. In addition, for Kratos, the file indicates the specific Kratos applications that are used in the workflow.

This information is passed to Spack which then retrieves the different package specs `package.py` files (one for each of the software components involved: PyCOMPSs, dislib and Kratos) from the Software Catalog. The variants that are needed and the target machine architecture are passed in the JSON file. In the example shown in figure 13, we are requesting to build a Singularity container image for an Intel SandyBridge architecture which is the architecture of the Nord3 supercomputer.

Once the image has been built, it is uploaded into an image repository for further reuse and Alien4cloud deploys it in the target system. The same procedure has been used to generate the image for the Intel Skylake architecture which is the one of the MareNostrum 4 supercomputer.

Besides the automated method provided by the eFlows4HPC stack, we have manually created and deployed a similar container image following standard container processes. In this case, we have created a Singularity image using the default deb packages of an Ubuntu distribution and the x86_64 pip packages for the PyCOMPSs, dislib and Kratos Multiphysics.

Experimental Results Figure 14 and Figure 15 show the execution times of the Pillar I workflow with the different container build processes and various core counts per task. These times are compared with the execution times using the native installation in bare metal.

Figure 14 shows the execution time comparison for the MareNostrum 4 cluster. We can see that our HPC ready approach reaches almost the same performance and scalability as the native installation. This is possible since the container creation process includes processor optimizations for the target architecture. In contrast, the generic containers are between 27% and 35% slower than the HPC ready containers. Similar results are obtained for the Nord3 cluster (depicted in Figure 15). In this case, the gain of using HPC ready containers is between 13% and 17%. The gain is more significant in the MN4 case because the architecture has some features (such as AVX instructions) not available in the Nord3 cluster.

Earth System Model workflow: eFlows4HPC Pillar II application

eFlows4HPC Pillar II develops innovative adaptive workflows for climate modelling. One of these workflows focuses on the development of the Earth System Model (ESM) by performing ensemble member simulation with AI-assisted member pruning. This workflow is one of the most challenging HPC use cases due to very high computational cost and additional challenges related to intensive I/O patterns, huge data volumes, and the necessity of not only data on HPC but also in-situ post-processing. The goal is to make a better use of computational and storage resources by performing a smart (AI-driven) pruning of ensemble members (and releasing resources accordingly) at runtime. The overall workflow is orchestrated with PyCOMPSs and uses Hecuba to store some intermediate simulation results. Hecuba is a set of tools and interfaces that implement a simple and efficient access to data stores for big data applications.

The workflow starts with an initialization step that determines the number of ensembles, initial conditions, the path to input datasets and definition of other execution parameters. The workflow bases its simulations on the Finite Element Sea Ice-Ocean Model (FESOM2) Danilov et al. (2017). FESOM2 is a multi-resolution ocean general circulation model that solves the equations of motion describing the ocean and sea ice using finite-element and finite-volume methods on unstructured computational grids.

The main execution step involves the execution of the model and simultaneously performs a periodic check of the state of the members. If the dynamical analysis recommends discarding members, the PyCOMPSs runtime manages their cancellation from the ensemble. All the outputs and files generated by the pruned members are discarded.

While the final version should include the different software components involved in the workflow, the results shown in this article are based on a container image that only consists of the FESOM2 simulator.

Image Creation and Deployment To create and deploy the images, we have followed the same procedure as for the Pillar I. In the workflow software requirements (spack.yml), we have just added FESOM2 as mentioned above. As FESOM2 depends on MPI, we have also included the MPI version installed in the machine in the JSON request. This will allow the Container Image Creation service to create a container with a compatible MPI version. Figure 16 shows the container creation request for the Nord3 supercomputer. A similar request is performed for MareNostrum 4 cluster with the same `intel-mpi` version but changing the processor architecture to Skylake.

Regarding the generic image, we have manually followed the instructions from the FESOM2 website to create a Docker image. It uses an Ubuntu distribution as base image, the FESOM2 dependencies are installed from Debian packages, and FESOM is installed from a source code installation with the generic x86_64 flags. This image has

Workflow Registry: https://github.com/eflows4hpc/workflow-registry
Software catalog: https://github.com/eflows4hpc/software-catalog
Hecuba website https://github.com/bsc-dd/hecuba
FESOM2 Docker installation: https://fesom2.readthedocs.io/en/latest/getting_started/getting_started.html#docker-based-installation

```
spack:
        specs:
                - compss
                - py-dislib
                - kratos apps=LinearSolversApplication,FluidDynamicsApplication,
  ↪ StructuralMechanicsApplication,
  ↪ ConvectionDiffusionApplication,RomApplication
```

Figure 12. Sample spack.yml file for the Pillar I workflow. This file lists the software components that will be installed in the Pillar I container.

```
{
    "machine": {
         "platform": "linux/amd64",
         "architecture": "sandybridge",
         "container_engine": "singularity"
  ↪
         },
    "workflow":"rom_pillar_I",
    "step_id" :"reduce_order_model"
}
```

Figure 13. Container configuration for Pillar I (JSON file)

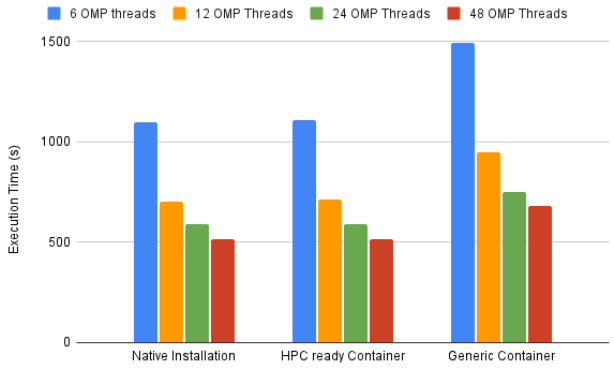

Figure 14. Execution times for the Pillar I workflow in the MN4 supercomputer with different core counts per task

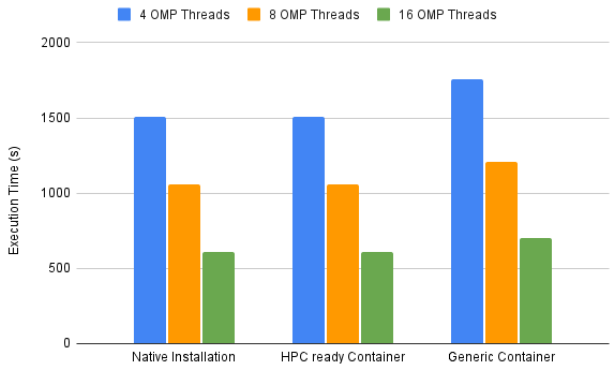

Figure 15. Execution times for the Pillar I workflow in the Nord3 supercomputer with different core counts per task

been converted to a singularity image and copied to the Nord3 and MareNostrum clusters.

Experimental Results Figure 17 and Figure 18 show the execution times of a 10-day simulation of the FESOM2

```
{
    "machine": {
         "platform": "linux/amd64",
         "architecture":
  ↪ "sandybridge",
         "container_engine":
  ↪ "singularity",
         "mpi": "intel-mpi@2018"
         },
    "workflow": "pillar_II",
    "step_id": "esm"
}

```

Figure 16. Container Configuration for Pillar II

model with the Core2 mesh and the containers generated with the two methodologies. The same simulation has been executed with 72, 144 and 288 MPI processes. These times are compared against the execution times with the native installation in bare metal.

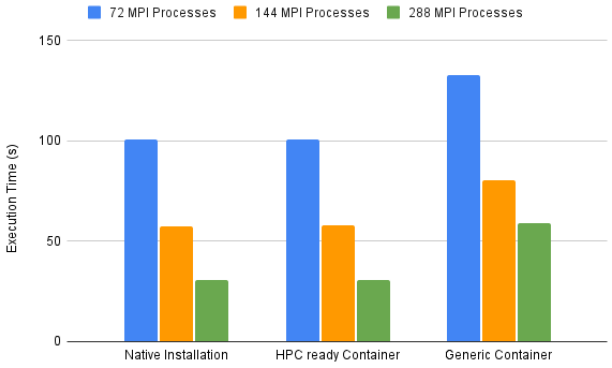

Figure 17. Execution times for the Pillar II workflow (FESOM2 model simulation) in the MN4 supercomputer with different number of MPI processes

Figure 17 shows the times for the MareNostrum 4 cluster. Our approach has a similar performance to the native compilation. In contrast, the generic container is from 30% worst for the 72 processes case to 90% worst for the 288 processes case. In the Nord3 executions (Figure 18), we have observed the generic container can be 115% slower than the native compilation. In this case, the difference in performance is due to two factors: processor optimizations and MPI usage. In the HPC-ready version, we could apply the processor optimizations at build time and select the MPI version, which allows us to bind the container to the MPI

host installation, supporting access to an efficient network fabric. The machines used in the evaluation have different capabilities. MN4 can run 48 MPI processes per node and its processor supports vector instructions. Nord3 can run 16 MPI processes and its processor does not support vector instructions. Since the execution is running in a reduced number of MN4 nodes, the communication does not have a big impact, and the gain is mainly achieved by the processor optimizations. However, in the case of Nord3, the application uses more nodes and communications are more important. Therefore, in this case, the gain is mainly due to the MPI efficiency.

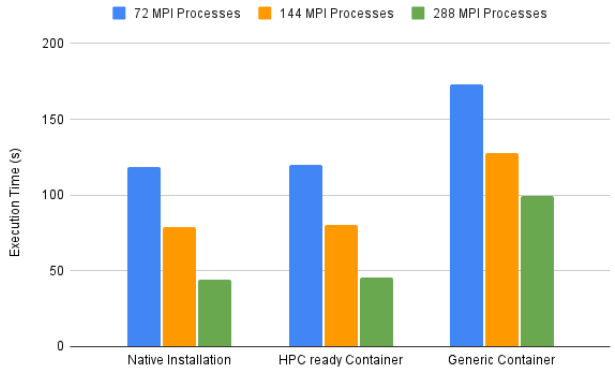

Figure 18. Execution times for the Pillar II workflow (FESOM2 model simulation) in the Nord3 supercomputer with different number of MPI processes

PTF/FTRT: eFlows4HPC Pillar III application

eFlows4HPC Pillar II develops workflows for urgent computing for natural hazards, more specifically for Earthquakes and their subsequent Tsunamis. The Pillar III Tsunami Workflow (PTF – Probabilistic Tsunami Forecast/FTRT - Faster Than Real Time) seeks to provide a forecast of tsunami impact following a large offshore or near-shore earthquake. The uncertainty in the source is dealt with by considering a (potentially very large) ensemble of earthquake scenarios. For each such scenario, an efficient numerical simulation needs to be performed in which the impact on the coastlines of interest is calculated. Just as numerical weather prediction generates a probabilistic forecast based on the outputs from multiple ensemble members, PTF calculates a probabilistic prediction of tsunami impact based on the outputs of the individual simulations and their scenario probabilities. The simulations in this workflow use the Tsunami-HySEA simulator Macías et al. (2017).

While the final version should include the different software components involved in the workflow, the results shown in this article are based on a container image that only includes the Tsunami-HySEA simulator.

Image Creation and Deployment To create and deploy the images, we have followed the same procedure as for Pillar I and II. In the workflow software requirements (spack.yml), we have just added Tsunami-HySEA as mentioned above. As Tsunami-HySEA depends on MPI and CUDA, we have also included the MPI and CUDA versions installed in the

```
{
    "machine": {
        "platform": "linux/ppc64le",
        "architecture": "power9le",
        "container_engine":
↪   "singularity",
        "mpi": "openmpi@4",
        "gpu": "cuda@10.2"
        },
    "workflow": "pillar_III",
    "step_id": "ftrt"
}
```

Figure 19. Container Configuration for Pillar III

machine in the JSON request. In this JSON request, we have changed the platform to linux/ppc64le and as architecture Power9, This will allow the Container Image Creation service to create a compatible container with the platform architecture and the available MPI and CUDA versions. Figure 19 shows the container creation request for the CTE-Power supercomputer.

For the generic container images, we have manually created a generic image for the `linux/ppc64le` platform using Docker buildx and the official NVIDIA CUDA 10.2 docker images for Ubuntu 18.04 (nvidia/cuda:10.2-devel-ubuntu18.04). On top of this base image, we have compiled the Tsunami-HySEA and its dependencies (OpenMPI and netCDF) with a generic compilation from sources. This was required since the CTE-Cluster has CUDA 10.2 and OpenMPI 4 installed, and the Ubuntu 18.04 deb packages only has OpenMPI 2 available. Therefore, there was a version mismatch and the image with OpenMPI 2 was not able to run in the CTE-Power. Finally, the build image has been converted to Singularity and copied to the CTE-Power cluster.

Experimental Results Figure 20 shows the execution time of the Tsunami-HySEA for the Mediterranean region using 1, 2 and 4 GPUs for the container images created with the two methodologies (HPC Ready and Generic) and the native compilation in bare metal. The multi-GPU executions are configured in a way that each Tsunami-HySEA process executes a scenario. In the case of 1 GPUs, a single scenario is simulated. In the case of 2 GPUs, two scenarios are simulated in parallel, and four for the case of 4 GPUs. For all the cases, we can see the three options are giving almost the same times. This is mainly because most of the computation is done in the GPU and the slight difference is due to the load of the problem to be executed.

Conclusions

With the evolution of the technology, HPC systems are every time larger, more complex and more heterogeneous. At the same time, application developers are also willing to leverage the huge performance that is offered to them and provide more complex applications that comprise different types of computations (i.e., traditional HPC modelling and simulation, data analytics, machine learning, etc.). However, very few efforts have been devoted to simplify the development, deployment and execution of complex

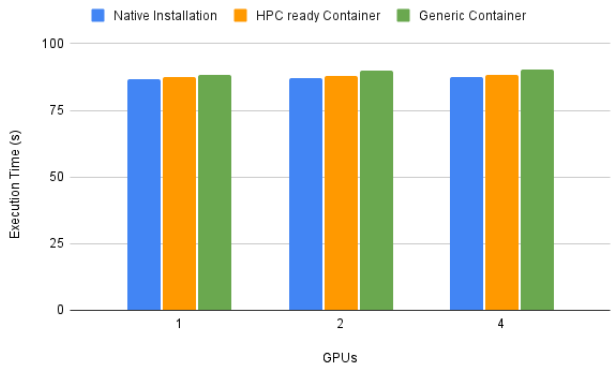

Figure 20. Execution times for the Pillar III workflow (T-HySEA model simulation) in the CTE-Power supercomputer with different GPUs

applications in HPC systems. Most approaches are based on ad-hoc solutions for each of the workflow applications.

The eFlows4HPC project focuses on providing generic methodologies and technologies that simplify the development, deployment and execution of complex workflows in HPC systems, and at the same time promoting the wider use and reuse of applications in HPC systems. Within the efforts in the project, this article describes the Container Image Creation (CIC) service, which aims at automatically building container images that leverage the feature of the target system. The service is based on the use of HPC builder systems (Spack) and container builder systems (Docker buildx). The service is evaluated in this article with applications from the three project Pillars in the areas of manufacturing, climate and urgent computing for natural hazards. The evaluation compares the specialized containers generated by the CIC service against generic containers and a native installation. The performance obtained demonstrated that the specialized containers have comparable performance to the native installation and up to 2× faster execution than the generic containers.

Acknowledgements

This work has been supported by the Spanish Government (PID2019-107255GB) and by MCIN/AEI /10.13039/501100011033 (CEX2021-001148-S), by Generalitat de Catalunya (contract 2021-SGR-00412), and by the European Commission's Horizon 2020 Framework program and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558 and by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR (project eFlows4HPC).

References

Boschert S, Bosco A, de Parga Regalado SA, Stabile G, Hernández JA and Bravo R (2021) D4. 2 design of the pillar i use cases URL https://eflows4hpc.eu/wp-content/uploads/2022/09/eFlows4HPC_D4.2-Design-of-the-Pillar-I-use_V1.0.pdf.

Dadvand P, Rossi R, Gil M, Martorell X, Cotela J, Juanpere E, Idelsohn SR and Oñate E (2013) Migration of a generic multi-physics framework to hpc environments. *Computers & Fluids* 80: 301–309.

Dadvand P, Rossi R and Oñate E (2010) An object-oriented environment for developing finite element codes for multi-disciplinary applications. *Archives of computational methods in engineering* 17(3): 253–297.

Danilov S, Sidorenko D, Wang Q and Jung T (2017) The finite-volume sea ice–ocean model (fesom2). *Geoscientific Model Development* 10(2): 765–789.

Ejarque J, Badia RM, Albertin L, Aloisio G, Baglione E, Becerra Y, Boschert S, Berlin JR, D'Anca A, Elia D, Exertier F, Fiore S, Flich J, Folch A, Gibbons SJ, Koldunov N, Lordan F, Lorito S, Løvholt F, Macías J, Marozzo F, Michelini A, Monterrubio-Velasco M, Pienkowska M, de la Puente J, Queralt A, Quintana-Ortí ES, Rodríguez JE, Romano F, Rossi R, Rybicki J, Kupczyk M, Selva J, Talia D, Tonini R, Trunfio P and Volpe M (2022) Enabling dynamic and intelligent workflows for hpc, data analytics, and ai convergence. *Future Generation Computer Systems* 134: 414–429. DOI:https://doi.org/10.1016/j.future.2022.04.014. URL https://www.sciencedirect.com/science/article/pii/S0167739X22001364.

Gamblin T, LeGendre M, Collette MR, Lee GL, Moody A, De Supinski BR and Futral S (2015) The spack package manager: bringing order to hpc software chaos. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. pp. 1–12.

Geimer M, Hoste K and McLay R (2014) Modern scientific software management using easybuild and lmod. In: *2014 First International Workshop on HPC User Support Tools*. IEEE, pp. 41–51.

Höb M and Kranzlmüller D (2020) Enabling easey deployment of containerized applications for future hpc systems. In: *International Conference on Computational Science*. Springer, pp. 206–219.

Kurtzer GM, Sochat V and Bauer MW (2017) Singularity: Scientific containers for mobility of compute. *PloS one* 12(5): e0177459.

Macías J, Castro MJ, Ortega S, Escalante C and González-Vida JM (2017) Performance benchmarking of tsunami-hysea model for nthmp's inundation mapping activities. *Pure and Applied Geophysics* 174(8): 3147–3183.

McMillan S (2018) Making containers easier with hpc container maker. In: *Proceedings of the SIGHPC Systems Professionals Workshop (HPCSYSPROS 2018), Dallas, TX, USA*, volume 10.

Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014(239): 2.

OASIS (2022) Topology and orchestration specification for cloud applications, version 2.0. URL http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

Olaya P, Kennedy D, Llamas R, Valera L, Vargas R, Lofstead J and Taufer M (2022) Building trust in earth science findings through data traceability and results explainability. *IEEE Transactions on Parallel and Distributed Systems* 34(2): 704–717.

Tejedor E and et al (2017) PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications (IJHPCA)* 31: 66–82. DOI:10.1177/1094342015594678.

Álvarez Cid-Fuentes J, Solà S, Álvarez P, Castro-Ginard A and Badia RM (2019) dislib: Large Scale High Performance Machine Learning in Python. In: *Proceedings of the 15th International Conference on eScience*. pp. 96–105.