Master in High Performance Computing

Containerization and deployment of weather models on HPC infrastructures

Supervisors:
Irina Davydenkova (ICTP)
Giuseppa Muscianisi (CINECA)

Candidate:
Massimo GISONNI

9th edition
2022–2023

Master in High Performance Computing

Abstract

In this thesis, we develop and demonstrate a robust and scalable workflow for building, deploying, and testing containerized weather models on the HPC clusters of the EuroHPC Joint Undertaking. The workflow is designed to support different cluster architectures, including x86-based systems (LEONARDO, GALILEO100, MELUXINA) and Cray-based systems (LUMI). Containerization is achieved using the Singularity platform, with container images built via Singularity definition files and optimized through multi-stage builds for lightweight and efficient images. The model-specific software stack is managed and deployed within the container using Spack, which enables the creation of customized environments tailored to the model's needs. We also highlight the necessary compatibility conditions required for the containerized model to fully leverage the underlying hardware. As a proof of concept, we successfully containerize the RAPS bundle of the IFS global weather model, developed by the European Centre for Medium-Range Weather Forecasts (ECMWF), which utilizes hybrid parallelism (MPI+OpenMP) and GPU acceleration. The containerized model is deployed and benchmarked across multiple clusters, with performance comparisons against native host installations. The results demonstrate that the containerized model achieves performance on par with a host-based execution, with no significant degradation, showcasing the scalability and efficiency of the containerized approach for HPC environments.

Acknowledgments

This thesis constitutes the final project for the Master in High Performance Computing [22], academic year 2022-23. The work was carried out at CINECA [3] within the framework of the Destination Earth initiative [9], under the agreement ECMWF/DESTINATIONEARTH/2022/DE_360 *signed between ECMWF [13] and CINECA as the contractor.*

I am grateful to the entire MHPC staff for introducing me to the fascinating world of HPC, to my alma mater SISSA for sponsoring my scholarship, and to the CINECA HPC staff for their support and expertise.

Thanks to my supervisors, Irina Davydenkova and Giuseppa Muscianisi, for guiding me throughout the master's program and this thesis, and to Thomas Geenen for supervising the project and reviewing this work.

I also want to thank all my MHPC classmates, who made this intense year a delightful one, with special mention to the boss Daniel Panea Lichtig and the amazing Luis Alfredo Nunez Meneses. Thanks to all my CINECA colleagues who contributed to this project, Spack master Francesco Cola, and my High Performance Forecasting teammates Gian Franco Marras, Fabio Di Sante and Matteo Ippoliti.

Contents

Introduction

Overview

In this thesis, we present a workflow for building, deploying, and testing containerized weather models on the HPC clusters of the EuroHPC Joint Undertaking [11]. The target clusters include systems based on x86 architectures, namely LEONARDO [7], GALILEO100 [5] and MELUXINA [21], as well as Cray architectures, LUMI [19].

The containerized weather models considered in this work include the *ECMWF Integrated Forecasting System (IFS)* and the *ECMWF Ocean Wave Model (ecWAM)*. The IFS model [14], developed and maintained by the *European Centre for Medium-Range Weather Forecasts (ECMWF)* [13], is one of the most advanced global weather models in the world. It operates as a coupled model, simulating the interactions between the atmosphere, ocean, land, and ice on a global scale. The IFS model also functions as a data assimilation system, integrating observations from a wide range of sources, including satellites, aircraft, and ground-based stations. The ecWAM model [12] is a component of the IFS system that describes the generation and evolution of wind-driven surface waves, capturing characteristics such as wave height, direction, and period.

The main advantage of software containerization in HPC clusters is the *portability* of container images across different hosts, removing the need to rebuild the software for each system. This capability is particularly valuable for managing software stack updates, which are often beyond the control of cluster users. When the host software stack changes, a container image can provide a stable and functional version of a model, ensuring continuity while allowing adaptation to new libraries and configurations. Additionally, containers enable users to test new compilers or runtime environments in isolation, anticipating future host system updates.

Building containers for HPC presents key challenges. Ensuring that the containerized applications perform without degradation compared to host-based installations requires correctly leveraging host-specific hardware, such as *InfiniBand* for MPI communication and *GPUs* for computational acceleration. Moreover, container builds must be *reproducible* to support CI/CD pipelines and maintain consistency across systems. Finally, the size of container images must be minimized to ensure *portability* and enable rapid deployment across clusters.

In this project, we develop a workflow based on *Singularity* and *Spack* to build container images for the two models presented above, IFS and ecWAM. Singularity [35] is the de facto container software in HPC environments, designed to run container images directly as user processes, ensuring compatibility with multi-user systems. Spack [31] is a package

management system tailored for efficiently handling software installations, dependencies, and configurations, automating the process and relieving users from the burden of manually managing dependency trees when deploying a software stack.

Singularity container builds are defined in *Singularity definition files*, while *Spack environments* are specified in *spack.yaml* files, which provide a means for templated, ad-hoc software deployment. Combining these two features offers a manageable and reproducible approach to software containerization: the container build is defined via a Singularity definition file, while the software stack required for the model use case is specified in a spack.yaml file and deployed within the container.

While this workflow ensures reproducibility in container builds, performance consistency between containerized and host-based builds must also be guaranteed. At runtime, we execute containerized software using a *Singularity hybrid model* approach [32], where MPI processes on the host interact with those inside the container to run parallel distributed jobs. We describe precise criteria to ensure no performance degradation, focusing on compatibility for MPI communications (MPI and UCX ABI compatibility [30, 37]) and GPU offloading (CUDA/ROCm compatibilty [1, 24]). To achieve this, we carefully select base images for the containers, such as those provided by the *NVIDIA HPC SDK* catalog for x86 systems [27].

The resulting containerized models were benchmarked on the target HPC clusters, demonstrating little to no performance degradation compared to host-based models. In several tests, the containerized models even outperformed the host versions.

Beyond the scope of this thesis, this work continues with efforts to test and deploy new containers as the models and software stacks evolve. Additionally, we work to integrate this approach into a CI/CD pipeline, enabling the rapid deployment of weather models for critical situations.

Structure of the Thesis

Chapter 1 introduces the primary tools and concepts used in this work. In Section 1.1, we explore the history and key features of containerization in HPC environments, while Section 1.2 presents the main mechanisms and commands for Spack and Singularity. The hardware and software characteristics of the GALILEO100, LEONARDO, MELUXINA, and LUMI clusters are detailed in Section 1.3.

Chapter 2 focuses on the containerization process for the specific use case of this project. Section 2.1 provides an overview of the two weather models under consideration, IFS and ecWAM, while Section 2.2 outlines the entire workflow for both x86 and Cray architectures, including snippets of the relevant spack.yaml and Singularity definition files.

In Chapter 3, the local configuration and performance evaluation of the containerized models on the different target clusters are presented.

Finally, Chapter 4 summarizes the project's achievements and discusses potential future directions.

Chapter 1

Overview of Software and Hardware

In this chapter, we introduce the key concepts and tools used in this project, focused on developing a consistent workflow for building container images compatible with different HPC clusters, including those with `x86` and `Cray` architectures. The two primary software tools employed are `Singularity` and `Spack`. `Singularity` is the de facto standard for running containers in HPC environments, while `Spack` is a versatile package manager that simplifies the deployment of software stacks within containerized systems.

Section 1.1 begins by exploring the concept of container images, including a historical overview in Section 1.1.1 and an introduction to their features in HPC contexts in Section 1.1.2. The main features and commands of `Singularity` are outlined in Section 1.2.1), with an emphasis on building container images, Section 1.2.1.1, and deployment on HPC infrastructures, Section 1.2.1.2. An introduction to `Spack` and its commands is given in Section 1.2.2. Finally, Section 1.3 details the software and hardware configurations of the target machines for this project: GALILEO100, LEONARDO, MELUXINA, and LUMI.

1.1 Containers

The development of containerization reflects a continuous effort to achieve efficient and secure isolation of processes in computing environments. Over the years, key innovations have emerged, each building on its predecessors, culminating in what is now a fundamental tool in HPC environments. In Section 1.1.1, we briefly recap the history of containers, while in Section 1.1.2, we analyze their integration into HPC clusters.

1.1.1 A Brief History of Containerization

1979: `chroot` – The Beginning of Process Isolation. The concept of isolating processes within restricted environments dates back to 1979, with the introduction of the `chroot` system call in *Unix Version 7*. The `chroot` operation modifies the apparent root directory (/) of a process and its child processes to a designated location in the filesystem hierarchy, confining them to a specific subset of the filesystem. However, `chroot` offered minimal isolation, as it did not extend beyond the filesystem. Processes within the `chroot` environment could still interact with system-wide resources like process IDs and network

interfaces. Additionally, it lacked resource control mechanisms and protections against malicious users breaking out of the restricted environment.

Despite these limitations, `chroot` laid the groundwork for more advanced methods of process isolation, introducing the concept of localized environments for processes, which would be expanded upon by subsequent innovations.

2000: FreeBSD Jails – Enhanced Sandboxing. Building on `chroot`, *FreeBSD Jails* introduced more robust process isolation in 2000. Jails extended `chroot` by creating isolated environments with their own network interfaces and IP addresses. They prevented cross-environment access and disallowed operations like raw socket access. In addition, Jails allowed administrators to limit CPU and memory usage, offering improved resource control. While Jails improved upon `chroot`, they lacked more advanced resource management and finer-grained isolation, limiting their flexibility and scalability compared to later container technologies.

2008: Linux Kernel Advancements – `cgroups` and Namespaces. The year 2008 marked a turning point with the introduction of two critical Linux kernel features: `cgroups` and Linux Namespaces.

- **`cgroups` (control groups):** A mechanism to isolate and manage resource usage (CPU, memory, disk I/O, network) for groups of processes.
- **Linux Namespaces:** A feature that partitions kernel resources, allowing processes in different namespaces to operate independently. For example, network namespaces isolated network interfaces, while process namespaces separated process IDs.

Together, `cgroups` and namespaces provided the necessary tools for process-level isolation and resource management, laying the technical foundation for modern containers.

2008: LXC (Linux Containers). In 2008, *LXC* emerged as the first widely-used tool to implement `cgroups` and namespaces. LXC enabled operating-system-level virtualization, where containers shared the host kernel but were isolated at the process level. It allowed running full Linux distributions within containers, providing a high degree of flexibility. LXC also leveraged `cgroups` and namespaces for effective resource management and isolation. However, LXC required manual configuration and lacked the user-friendly tools that Docker later introduced.

2013: Docker – Streamlining the Containerization Process. The modern containerization revolution began with the release of *Docker* in 2013 by dotCloud Inc. Docker, initially based on LXC, introduced several key innovations that reshaped the way software was packaged, distributed, and deployed:

- **Docker Images:** Standardized, portable packages containing an application and its dependencies.

- **Dockerfile:** A configuration file for building images, simplifying container creation.
- **Image Registry:** A centralized repository for storing and sharing images.

Docker's emphasis on ease of use and developer experience transformed containers from a tool for Linux administrators into a mainstream technology. By decoupling applications from infrastructure, Docker enabled faster development cycles, consistent deployments, and scalability across environments.

2016: Singularity – Containerization for HPC Environments. While Docker revolutionized containerization for general-purpose computing, its design posed challenges in High-Performance Computing (HPC) environments. In response, *Singularity* was introduced in **2016** by Gregory Kurtzer [17], specifically targeting the unique requirements of HPC systems. Singularity quickly became the go-to containerization platform for HPC, addressing key limitations of Docker in this domain:

- **User Privileges:** Singularity allows non-privileged users to run containers, eliminating the need for root privileges and improving security in multi-user HPC systems.
- **HPC Workflow Integration:** Singularity integrates seamlessly with batch schedulers like SLURM, PBS, and LSF, enabling efficient job submissions.
- **Filesystem Accessibility:** Singularity containers operate within the user's filesystem namespace, simplifying access to shared resources.
- **Single Binary Design:** Singularity containers are distributed as single, immutable files (.sif format), ensuring portability and tamper-proof containers.
- **MPI and GPU Support:** Singularity supports MPI workloads and GPU passthrough, critical for HPC applications.

Singularity's HPC-oriented features and its lightweight runtime led to its rapid adoption in research and academic institutions. Unlike Docker, Singularity minimizes overhead and maintains compatibility with many existing container images, making it an efficient choice for complex scientific software. In Section 1.2.1, we will delve more deeply into *Singularity*, presenting the key main commands and functionalities.

In 2021, Singularity was rebranded as *Apptainer* under the Linux Foundation. Apptainer maintains compatibility with legacy Singularity containers and benefits from open-source governance for continued development and broader community contributions.

1.1.2 Container Functionalities and HPC Integration

Modern containers achieve process isolation and resource management by virtualizing at the operating system level. Unlike virtual machines (VMs), which emulate hardware and encapsulate an entire operating system, containers share the host kernel while isolating user spaces. This approach eliminates the need for redundant operating system components, optimizing resource usage and enabling lightweight, scalable deployments.

- **Virtual Machines:** VMs operate on a hypervisor, with each instance running its own kernel and complete OS stack. This model ensures strong isolation and compatibility but at the expense of significant resource overhead, slower boot times, and reduced flexibility.

- **Containers:** Containers virtualize user spaces, leveraging the shared host kernel to create isolated environments for applications. Shared libraries and runtime dependencies are read-only, while each container maintains a writable layer for its specific needs. This design minimizes resource usage and startup latency, making containers highly suitable for dynamic and scalable workloads.

This OS-level virtualization model not only reduces redundancy but also enhances efficiency, positioning containers as a versatile solution for modern application deployment and management. To achieve these advantages, containers rely on core kernel features:

- **Kernel Sharing and API Usage:** Containers share the host kernel, avoiding the overhead of running separate guest operating systems. The host kernel provides APIs that facilitate the creation of isolated user spaces, enabling each container to function as an independent environment while maintaining lightweight operations.

- **System Calls and Namespaces:** Namespaces provide process-level isolation by partitioning system resources such as process IDs, file descriptors, network interfaces, and mount points. For instance, a containerized application perceives its PID namespace as unique, abstracting away the shared nature of the underlying kernel. This partitioning ensures that containers operate independently, unaware of other containers or the host system.

- **Resource Governance via Control Groups:** `cgroups` enforce resource usage limits for groups of processes, regulating CPU, memory, disk I/O, and network bandwidth. A `cgroup` is essentially a hierarchy with processes as nodes. It associates resource quotas or limits with processes and propagates these constraints during runtime. This ensures predictable performance and fairness in resource sharing, particularly critical in multi-tenant environments.

Containers have gained popularity HPC due to their ability to provide lightweight, scalable, and efficient solutions for running applications. In the next paragraphs we briefly describe their integration with respect to distributed computing, multi-user environments, and specialized hardware usage.

Resource Managers. Resource managers such as `SLURM`, `PBS`, and `LSF` handle job scheduling and resource allocation across nodes in a cluster. Containers integrate into these systems by encapsulating applications and respecting the resource allocation boundaries set by the scheduler. For example, with SLURM, the `srun` command can be used to launch containerized jobs. Container runtimes like *Singularity* ensure that containerized tasks follow the scheduling policies and resource limits defined by the cluster.

Filesystem Accessibility. HPC systems often rely on high-throughput parallel file systems such as `Lustre`, `BeeGFS`, and `GPFS` for data access. Containers need to interact with these shared file systems while maintaining user permissions and directory structures. Container runtimes map the host's filesystem namespaces into the container, allowing applications to access shared resources natively. This integration ensures compatibility with tools and libraries that rely on specific filesystem paths.

Multi-User Environment Security. Security in shared HPC clusters is critical. Containers run within user-specific namespaces, preventing privilege escalation and ensuring that containerized applications do not gain root access. Runtimes like *Singularity* allow containers to be executed without requiring elevated privileges, maintaining robust isolation. This approach ensures compatibility with multi-user policies, essential for maintaining a secure environment.

MPI and Distributed Computing. Many HPC applications rely on the *Message Passing Interface (MPI)* for parallel processing across multiple nodes. Running MPI applications in containers requires careful configuration of network and library settings to ensure compatibility with the host's infrastructure. Containers can be set up to inherit the host's high-performance interconnects, such as InfiniBand, enabling seamless communication between nodes. See also Section 1.2.1.2 for a more detailed description of this framework with respect to Singularity-

Hardware Acceleration HPC workloads often demand access to specialized hardware like GPUs. Containers support hardware acceleration through integration with tools like `NVIDIA-Docker` and native CUDA support in *Singularity*. These mechanisms allow containerized applications to access GPUs and other hardware resources by binding device files (e.g., `/dev/nvidia*`) into the container's namespace, enabling the use of hardware acceleration without modifying the application code.

1.2 Project Main Tools: Singularity and Spack

In this project, we develop a workflow based on `Singularity` and `Spack` to build container images tailored to specific use cases. In a nutshell, we select a suitable base image and use Singularity definition files to build the container image, installing the necessary dependencies via Spack. `Singularity` ensures reproducibility, while `Spack` addresses legacy requirements (version freezing) and simplifies the installation process.

In Section 1.2.1, we provide an overview of the functionalities of the Singularity software, introducing the main concepts and commands for the build and execution phases, Sections 1.2.1.1 and 1.2.1.2 respectively. In Section 1.2.2, we introduce the Spack package management system, which provides a fast way to install complex software stacks.

In Chapter 2, we will present examples of complete Singularity definition files and Spack environment files used in the project.

1.2.1 Singularity

Singularity is the *go-to* container runtime for usage in HPC environments. Unlike Docker, Singularity does not rely on a background daemon process; containers run directly as user processes, making it compatible with HPC resource managers like SLURM and PBS.

1.2.1.1 Building a Singularity Image

Building a Singularity container image requires root privileges, meaning one must be able to `sudo` on the building host. However, Singularity offers options to circumvent this limitation in multiuser environments like HPC clusters: `fakeroot` and `proot`. `fakeroot` simulates root privileges for file ownership and permission operations, without actual root access, while `proot` simulates a root filesystem by intercepting system calls to create isolated environments for running applications or building software, all without requiring root access. Still, both of these options must be enabled by the system administrators and may not always be available.

The architecture of the system where the container is built must match the target machine's architecture. For example, a container built on an `x86_64` system cannot run on a `Cray` ARM-based system unless specific measures, such as cross-compilation or multi-architecture support, are taken. This is because *Singularity containers rely on the host system's kernel and hardware architecture* rather than emulating them.

Singularity images come in two main types: `sandbox` and `SIF (Singularity Image Format)`. A *sandbox* is a writable container that can be modified during the build process. It is essentially a directory on the host system where the container's filesystem is created and can be interactively modified. A *SIF* is a read-only, compressed image that contains the complete filesystem of the container. It is portable and ensures reproducibility, as the contents are fixed and cannot be modified once created.

The simplest way to build a container image via Singularity is to bootstrap from the Docker repository [10] using the following commands

```
singularity build image.sif docker://ubuntu:20.04
singularity build --sandbox my_sandbox.img docker://ubuntu:20.04
```

The command *must be run as sudo*. The build via `fakeroot` is handled appending the corresponding flag to the `build` command.

```
singularity build --fakeroot /path/to/image.sif /path/to/def_file.def
```

A *SIF* image can be converted to a *sandbox* and viceversa via

```
singularity build --sandbox /path/to/sandbox /path/to/image.sif
singularity build /path/to/image.sif /path/to/sandbox
```

Another way to build container image is via a **Singularity definition file** (also known as a "recipe"). A Singularity definition file is a script that describes how to build a container. It defines the base operating system, installation commands, environment variables, and additional configuration for the target container image. A typical definition file consists of several sections.

The `Header` defines the base image to be used for the container. The `Bootstrap` image can either be drawn from the Docker Hub repository,

```
Bootstrap: docker
From: ubuntu:22.04
```

or a local SIF image can be specified as bootstrap:

```
Bootstrap: localimage
From: /path/to/localimage.sif
```

The `%files` section copies files or directories from the host system into the container during the build process. e.g.

```
%files
    /host/path/to/file          /container/path/to/file
    /host/path/to/directory/*   /container/path/to/directory
```

Notice the wildcard `*` being used when copying directories. The target directory in the container is automatically created if not already present.
The `%post` section is the core of the definition file, containing commands that run during the build process. Packages can be installed, environment variables set, or the container's filesystem modified:

```
%post
    apt-get update && apt-get install -y python3
    mkdir -p /mysoftware
    ./install_script.sh
```

The `%environment` section specifies variables that will be available when the container runs:

```
%environment
    export MY_VAR=this_value
```

The `%runscript` section contains the default execution script that runs when the container is invoked:

```
%runscript
    echo "Running container"
    python3 /app/my_script.py
```

Other sections available are `%startscript`, `%test`, `%labels`, `%help`; see also the official documentation [34].

To build an image from a definition file, the following command is used:

```
singularity build /path/to/output_image.sif /path/to/def_file.def
```

Another interesting feature of Singularity definition files is the possibility to define `multistage builds` or matrioska builds. A multistage build allows to separate the build and deployment stages into distinct phases. This method offers the advantages of keeping the final image lightweight and optimized for runtime, as only the necessary files and dependencies are included in the final stage, while the build tools and temporary files are excluded. It is particularly useful in containerized workflows where the build and execution environments are different, or when there is a need to optimize the size of the final image. It also supplies for the lack of a cache mechanisms for Singularity builds, opposed to the one available with Docker.

In a multistage build, the bootstrap image in the second (or later) stage can either be the same as the one used in the first stage or a different one, depending on the compatibility and requirements of the application being deployed. Below is a minimal example of a multistage Singularity build, where the first stage handles the build and the second stage is focused on deployment.

```
Bootstrap: docker
From: ubuntu:20.04
Stage: build

%post
    apt-get update
    ./install_software.sh   --builddir=/build

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Stage: deploy
Bootstrap: docker
From: debian:bullseye

%files from build
    /build/bin/my_binary    /deploy/my_binary
```

Listing 1.1: Singularity Multistage Build Example

Singularity definition files automate the creation of Singularity container images, enabling the precise definition of reproducible environments. We provide full definition files for the project use case in Chapter 2, Listings 2.2 and 2.4.

1.2.1.2 Execution of a Singularity Application

Applications encapsulated within Singularity containers can be executed directly with commands tailored for seamless integration into HPC workflows. The foundational command, `singularity exec`, runs a specified binary or script from within the container:

```
singularity exec my_container.sif /path/to/binary/inside/container
```

This executes the binary located at the specified path within the containerized environment. For containers with a default application defined in their metadata, `singularity run` can be used to launch the predefined entry point directly.

Filesystem Binding and Mounting. While a Singularity container provides a standalone runtime environment, many workflows require access to external files or directories, either to read input data or to write results. *Binding* refers to the process of making host system paths available inside the container at runtime.

In its default configuration, Singularity automatically mounts several key paths from the host into the container, such as `/proc`, `/sys`, `/dev`, `/tmp`, the user's home directory, `$HOME`, and the current working directory, `$PWD`. These ensure compatibility with system tools and user data.

For custom bindings, users can specify additional paths with the `-B` or `--bind` option. Bind specifications follow the format `src[:dest[:opts]]`, where `src` is the host path, `dest` is the container path (defaulting to `src` if omitted), and `opts` are optional mount options. For example:

```
singularity exec -B /data:/mnt my_container.sif /bin/bash
```

This command mounts the host directory `/data` to `/mnt` inside the container. Alternatively, bindings can be pre-defined using the `SINGULARITY_BIND` environment variable, e.g.:

```
export SINGULARITY_BIND=/host/path:/container/path
```

For immutable SIF images, bindings are essential for achieving write privileges, as they lack the inherent write capabilities of sandbox directories.

Environment Variables. Singularity enables fine-grained control over environment variables inside the container. By default, variables from the host environment are passed into the container, but certain variables, such as `PATH`, `LD_LIBRARY_PATH`, and `PS1`, are modified to ensure compatibility with the containerized environment.

To override or define new environment variables specifically for the container, Singularity offers several mechanisms. For example, variables can be passed directly at runtime using the `--env` option, as in:

```
singularity exec --env MY_VAR=value my_container.sif /bin/bash
```

Alternatively, the `SINGULARITYENV_` prefix can be used to predefine environment variables in the host before execution. For instance, setting `export SINGULARITYENV_MY_VAR=value` in the host environment ensures that `MY_VAR=value` is accessible inside the container during runtime. This approach is particularly useful for passing multiple variables without modifying runtime commands.

For cases where numerous variables need to be defined, a text file containing key-value pairs can be used in conjunction with the `--env-file` option, allowing streamlined management of container environments.

For path-related variables, additional paths can be appended or prepended to the container's default `PATH` using the `SINGULARITY_APPEND_PATH` and `SINGULARITY_PREPEND_PATH` variables. Libraries specified in the `LD_LIBRARY_PATH` can be handled, as above, via the `SINGULARITYENV_LD_PREPEND_PATH`.

Finally, during the build process, two host environment variables can manage temporary data and cache locations. `SINGULARITY_CACHEDIR` specifies a directory to cache data such

as Docker layers or metadata, while `SINGULARITY_TMPDIR` defines the directory used for temporary files when creating a container image, such as during the build of the squashfs filesystem.

Integration with MPI Workloads. Singularity supports the execution of MPI workloads through two main integration approaches: the `hybrid model` and the `bind model`.

In the Singularity MPI `hybrid model` process outside of the container works in tandem with the MPI inside the container and the containerized MPI code to instantiate a parallel distributed job [32]. To achieve this, `MPI ABI` *(Application Binary Interface) compatibility* between the host and the container is a necessary condition. For performance equivalence at runtime, the `UCX` *(Unified Communication X)* libraries in both the host and container must also be ABI compatible. Backward compatibility for UCX is consistently maintained [37], ensuring that an MPI installation inside the container, built with a UCX version no newer than the one installed on the target machine, avoids performance degradation. Importantly, the hybrid model does not require additional network drivers (e.g., InfiniBand) to be included in the container, provided the containerized MPI stack includes support for OFED (OpenFabrics Enterprise Distribution) libraries.

The `bind model` leverages the host system's MPI libraries directly, minimizing compatibility issues between the containerized application and the host interconnect hardware. In this configuration, only the application and its dependencies are encapsulated within the container, while the host's MPI stack is mounted into the container at runtime. This is achieved using the `--bind` option to map the host's MPI library paths to locations within the container. A typical command might be:

```
mpirun -np 4 singularity exec --bind /host/mpi:/container/mpi
  mpi_container.sif /path/to/mpi_binary
```

In this case, the containerized application dynamically links to the host's MPI libraries, requiring correct paths to be exposed inside the container. The `LD_LIBRARY_PATH` environment variable must be updated accordingly to ensure the containerized application resolves the mounted libraries correctly.

Hardware Acceleration for GPUs. Singularity supports GPU-accelerated workloads by integrating directly with the host's GPU hardware. For NVIDIA GPUs, the `--nv` flag automatically mounts necessary libraries and device files from the host into the container. For example, via the command:

```
singularity exec --nv my_container.sif nvidia-smi
```

libraries like `libcuda.so` are automatically binded into the container, as well as device nodes under `/dev/nvidia*`. Similarly, the `--rocm` flag facilitates AMD GPU integration, binding ROCm libraries and device files:

```
singularity exec --rocm my_container.sif rocminfo
```

These options ensure that GPU-enabled applications can run seamlessly without requiring additional container configurations.

1.2.2 Spack

`Spack` is a package management system designed for HPC environments, enabling users to efficiently handle software installations, dependencies, and configurations. It leverages a modular approach, where software packages are defined using *Python-based recipes*, which specify the source code, dependencies, and configuration options. These recipes are stored in *repositories*, allowing Spack to automate the installation and configuration process while providing flexibility to manage complex environments with multiple versions, configurations, and hardware setups.

Spack simplifies software installation through the `spack install` command, offering fine-grained control over package versions, compilers, and optional features. For instance, the command:

```
spack install hdf5@1.12.2 %gcc@10.3.0 +mpi -cxx ^zlib@1.2.11
```

installs version 1.12.2 of the HDF5 library, specifying GCC version 10.3.0 as the compiler, enabling MPI support for parallel I/O and disabling C++ bindings. It also specifies the installation of `zlib` version 1.2.11 as a dependency. The `+feature` symbol includes variants, and `-feature` excludes them. The `@version` selects the desired version of the software, while `%compiler` indicates which compiler to use. The caret (`^`) is used to specify a particular version of a dependency to ensure compatibility with the required library version.

When resolving dependencies, Spack constructs an internal dependency graph to ensure compatibility among all required packages and their versions. For example, enabling `+mpi` for HDF5 might necessitate installing an MPI implementation, which Spack handles automatically. It is worth noting that the complexity of dependency resolution is an NP-complete problem [8].

Each software installation in Spack is uniquely identified by a *hash code*, derived from the software's configuration, dependencies, and build settings. This ensures that even installations of the same package can coexist with distinct configurations. For example, building `hdf5@1.12.2` with MPI support produces a different hash from a non-MPI build, maintaining clear separation and reproducibility across environments.

Spack supports different *installation scopes*: system-wide, user-specific, and environment-specific. In the environment scope, users can create isolated environments to maintain specific software stacks for different projects. These environments allow for reproducibility and easy management of software configurations.

Spack environments Spack environments can be managed using the `spack env` command, which allows for the creation and activation of isolated environments to manage package installations. An environment can be created and activated via:

```
spack env create myenv
spack env activate myenv
```

Once the environment is active, the configuration is defined in a `spack.yaml` file, which is automatically created. This file contains several sections.

The `config` section specifies paths for different aspects of the environment. For example:

```
config:
  source_cache: $env/source_cache
  install_tree: $env/install
  build_stage:
    - $env/build/stage
    - $tempdir/$user/spack-stage
```

Here, `source_cache` defines where the source code for downloaded packages will be cached, `install_tree` specifies the location of installed packages, and `build_stage` defines paths for the build stage.

The `compilers` section lists the compilers available in the environment. It can be manually configured or automatically populated using `spack compiler find`, which detects compilers available on the host system. For instance:

```
compilers:
- compiler:
    spec: gcc@11.4.0
    paths:
      cc: /usr/bin/gcc
      cxx: /usr/bin/g++
      f77: /usr/bin/gfortran
      fc: /usr/bin/gfortran
    operating_system: ubuntu22.04
    target: x86-64
```

The `packages` section lists software already installed on the host. It can be populated automatically by running `spack find`, which detects installed packages based on the `$PATH` variable:

```
packages:
  mpich:
    paths:
      mpich: /opt/mpich
    buildable: false
```

This section ensures that Spack uses the correct version of a package that is already installed on the system, avoiding redundant installations.

The `specs` section in `spack.yaml` defines the packages that need to be installed in the environment. For example:

```
specs:
- cmake@3.27
- osu-micro-benchmarks@8.1
```

specifies that the environment should install `cmake` version 3.27 and `osu-micro-benchmarks` version 8.1.

The dependencies for the packages in `specs`, compiled with compilers specified in `compilers` and eventually depending on libraries in `packages`, are automatically resolved

launching `spack concretize`. Once the dependency tree has been created, `spack install` installs the necessary packages with their correct configurations.

In our use case, we deploy `Spack` inside the container images, and compile a `spack.yaml` file specific to the needed model software stack. Examples of such files can be found in Chapter 2, Listings 2.1 and 2.3.

1.3 Target Clusters

Following our exploration of the main software tools employed in this project, this final introductory section provides an overview of the hardware platforms utilized for running the containerized models.

The benchmarks were conducted on four distinct HPC clusters: GALILEO100 (G100), LEONARDO, MELUXINA, and LUMI. Among these, the first three employ an `x86_64` architecture, while LUMI is endowed with a `Cray` architecture. A performance evaluation of the containerized models across these systems is presented in Chapter 3.

LEONARDO, MELUXINA, and LUMI are part of the *EuroHPC Joint Undertaking (JU)*, a European initiative aimed at developing and deploying high-performance computing infrastructure to support scientific research and industrial innovation in Europe [11].

Basic software In Table 1.1 we list the main parts of the machines software stack: the operative system, the compilers (used in the project), the software and drivers relative to GPU accelerators. The versions here reported are those relative to the time of testings of this project. For the current state of the art, refer to the user guides of the different clusters, [5, 7, 19, 21]

Table 1.1: Supercomputer Software Environments

System	Architecture	Operating System	Compilers	GPU Libraries and Drivers
GALILEO100	x86_64	CentOS@8.3	nvhpc@22.3	Cuda@11.5, 470.42.01
LEONARDO	x86_64	Red Hat@8.7	nvhpc@23.1	Cuda@[11.8,12.1], 530.20.02
MELUXINA	x86_64	Rocky Linux@8.8	nvhpc@23.7	Cuda@[11.7,@12.3], 535.104.12
LUMI	Cray	SLES15 SP5	cce@17.0.1	Rocm@6.3.6

1.3.1 GALILEO100 Specifics

GALILEO100 is hosted at the CINECA headquarters in Casalecchio di Reno, Italy. It is an updated version of the predecessor system, GALILEO, and was deployed in September 2021. GALILEO100 also hosts 77 servers for cloud computing. A summary of its architecture is provided in Table 1.2, see also [4].

1.3.2 LEONARDO Specifics

LEONARDO is a pre-exascale Tier-0 EuroHPC supercomputer hosted by CINECA, located in the Bologna Technopole, Italy. It is currently ranked 9th on the TOP500 list [36], with an Rmax of 241 PFlop/s and an Rpeak of 306 PFlop/s. LEONARDO is divided into two partitions:

- The `booster` partition, which consists of BullSequana XH2135 supercomputer nodes, each equipped with four NVIDIA Tensor Core GPUs and a single Intel CPU.
- The `DCGP (Data-Centric General-Purpose` partition, which features X2140 CPU-only blades based on Intel Sapphire Rapids processors.

The specifics of LEONARDO are summarized in Table 1.3, see also [6].

1.3.3 MELUXINA Specifics

MeluXina is a high-performance computing cluster hosted in Luxembourg data centers operated by LuxConnect. It was launched in June 2021 by LuxProvide and is built on the EVIDEN BullSequana XH2000 platform. Currently ranked 112th on the TOP500 list, MeluXina achieves an Rmax of 10.52 PFlop/s and an Rpeak of 15.29 PFlop/s. Additional details on its architecture and capabilities are available in Table 1.4, see also [20].

1.3.4 LUMI Specifics

LUMI (Large Unified Modern Infrastructure) is a a pre-exascale Tier-0 EuroHPC supercompute hosted at the CSC data center in Kajaani, Finland. It was inaugurated in June 2022 and is currently ranked 8th on the TOP500 list, achieving an Rmax of 379.70 PFlop/s and an Rpeak of 531.51 PFlop/s. LUMI is supplied by Hewlett Packard Enterprise (HPE) and features an HPE Cray EX supercomputer equipped with next-generation 64-core AMD EPYC CPUs and AMD Radeon Instinct GPUs. The system is divided into two primary partitions: the `LUMI-C` CPU partition and the `LUMI-G` GPU partition. Further details on LUMI's architecture and capabilities can be found in Table 1.5, see also [18].

Table 1.2: GALILEO100 Supercomputer Specifics

GALILEO100	
Model	Dual-Socket Dell PowerEdge
Nodes	528 computing nodes 348 standard nodes (480 GB SSD) 180 data processing nodes (2 TB SSD, 3 TB Intel Optane) 36 GPU nodes (100 GbE Infiniband, 2 TB SSD) 77 computing servers, 768 GB RAM (100 GbE interconnection)
Processors	2 x CPU x86 24 cores Intel Xeon Platinum 8276-8276L, 2.4 GHz
Accelerators	2 x NVIDIA V100 GPUs PCIe3 32 GB RAM on 36 Viz nodes
RAM	384 GB per node (+ 3.0 TB Optane on 180 fat nodes)
Peak Performance	About 2 PFlop/s
Internal Network	Mellanox Infiniband 100 GbE
Storage	20 PB of active storage (accessible from cloud and HPC nodes) 5 PB of fast storage for HPC system 1 PB Ceph storage for Cloud (full NVMe/SSD)

Table 1.3: Leonardo Supercomputer Specifics

LEONARDO		
Partition	**DCGP**	**Booster**
Model	Atos BullSequana X2140 three-node CPU blade	Atos BullSequana X2135 "Da Vinci" single-node GPU blade
Nodes	1536	3456
Processors	2 sockets, 56 cores Intel Sapphire Rapids 2x Intel Xeon Platinum 8480p 2.00 GHz, TDP 350W	1 socket, 32 cores Intel Ice Lake 1x Intel Xeon Platinum 8358 2.60 GHz, TDP 250 W
Accelerators	-	4 x NVIDIA Ampere GPUs/node, 64 GB HBM2e NVLink 3.0 (200 GB/s)
RAM	512 GB (16x32 GB) DDR5 4800 MHz	512 GB (8x64 GB) DDR4 3200 MHz
Peak Performance	9 PFlop/s	About 309 PFlop/s
Network Topology	DragonFly+ 200 Gbps (NVIDIA Mellanox Infiniband HDR)	
Internal Network	Single port HDR100 per node	2 x dual port HDR100 per node
Storage	137.6 PB (DDN ES7990X, Hard Drive Disks, Capacity Tier) 5.7 PB (DDN ES400NVX2, Solid State Drives, Fast Tier)	

Table 1.4: MELUXINA Supercomputer Specifics

MELUXINA		
Partition	**CPU**	**GPU**
Model	EVIDEN BullSequana XH2000	
Nodes	573	200
Processors	Dual socket AMD EPYC 7H12 64-Core Processor, 2.60 GHz	Dual socket AMD EPYC 7452 64-Core Processor, 2.35 GHz
Accelerator	-	4 x NVIDIA A100 GPUs/node 40 GB HBM NVLink3.0 (1,555 GB/s memory bandwidth)
RAM	256 GB per node	256 GB per node
Peak Performance	About 15 PFlop/s	
Internal Network	InfiniBand (IB) HDR 200 Gb/s high-speed fabric	
Storage	24.3 PB (0.5 PB Scratch, 12 PB Project, 6.7 PB Backup, 5 PB Archive)	

Table 1.5: LUMI Supercomputer Specifics

LUMI		
Partition	**CPU**	**GPU**
Model	HPE Cray EX Supercomputer	
Nodes	2048	2978
Processors	Dual socket AMD EPYC 7763 Zen 3, 2.45 GHz (base), 3.5 GHz (boost)	Single socket AMD EPYC 7A53 Trento Zen 3, 3.5 GHz
Accelerator	-	4 x AMD MI250X GPUs/node, 128 GB HBM per module, 220 Compute Units (CUs)/module
RAM	256 GiB per node	512 GiB per node
Peak Performance	379 PFlop/s (Rmax)	
Internal Network	HPE Slingshot 11, 200 Gb/s high-speed interconnect	
Storage	117 PB Lustre file system	

Chapter 2

Containerization of the IFS Model

In this chapter, we outline the workflow developed in this project for containerizing weather models on different target machines. Notably, we demonstrate how the *same* container images can be built and run on different clusters with identical architectures.

In Section 2.1, we introduce the IFS and ecWAM models and define the target clusters for deploying each of them. In Section 2.2, we present the general workflow developed in this project, followed by a detailed discussion of the `Singularity definition file` and `spack.yaml` files for the `x86` and `Cray`-tailored container images in Sections 2.2.1 and 2.2.2, respectively.

2.1 The IFS and ecWAM models

Description of the models The *ECMWF Integrated Forecasting System (IFS)* [14] is a comprehensive Earth system model developed and maintained by the *European Centre for Medium-Range Weather Forecasts (ECMWF)*. It is one of the most advanced and widely used weather forecasting systems in the world. The IFS produces a broad spectrum of forecasts, ranging from short-range weather forecasts spanning a few days to long-range climate projections covering many years. As a global model, the IFS encompasses the entire Earth and operates as a coupled model, simulating interactions between the atmosphere, ocean, land, and ice. The IFS model also functions as a data assimilation system, enabling the incorporation of observations from various sources, such as satellites, aircraft, and ground stations, into its forecasts, thereby enhancing forecast accuracy.

One of the component of the IFS model is the *ECMWF Ocean Wave Model (ecWAM)* [12], which describes the development and evolution of wind-generated surface waves, including their height, direction, and period. It primarily focuses on ocean wave forecasting. However, it's important to note that ecWAM does not model the ocean dynamics itself; instead, the dynamical modeling of the ocean is typically performed by a separate ocean model such as NEMO *Nucleus for European Modelling of the Ocean*, [23].

Models target clusters The ecWAM model, being fairly easier to handle than its parent model IFS, was chosen as the first software to be containerized in the project. As explained

in the next Section 2.2, we built a containerized version of ecWAM for all the three x86 architectures target machines, LEONARDO, GALILEO100 and MELUXINA.

With respect to the IFS model, in this project we specifically work with the `RAPS` bundles of the 48r1 and 49r1 cycle of IFS [14]. RAPS stands for *Real Applications on Parallel Systems*, and it is a build of the model designed for benchmarking and testing purposes. The RAPS bundle includes the code for the dynamics and physics of the model, while the IFS data assimilation component is not included. Instead, benchmarks and tests are performed using synthetic data. In the following sections, we will use the terms RAPS and IFS interchangeably to refer to the use case model.

The RAPS 48r1 bundle includes OpenMPI+OpenMP parallelism as well as GPU offloading, while the 49r1 bundle does not contain any GPU offloading. The 48r1 bundle was containerized and tested for the LEONARDO and MELUXINA clusters, but not for GALILEO100 as it is not provided with a GPU partition. On the other side, for LUMI, we were forced to deploy the 49r1 bundle (hence *without* GPU offloading) due to a recent update of the cluster software stack which made the 48r1 bundle obsolete.

2.2 Description of the Building Process

In this section, we describe the containerization of the IFS and ecWAM models for the target x86 architecture (LEONARDO, GALILEO100, MELUXINA) as well as Cray (LUMI). In both cases, the workflow essentially reduces to three steps:

1. Bootstrap from a container image compatible with the target machine,
2. Install the needed dependencies for the model using Spack,
3. Compile the model and turn the steps into a Singularity definition file.

Although one could start from a mostly empty container image with just the operating system and manually install the necessary software stack (including compilers and MPI), it is more convenient to start from an image that already contains at least the compilers and an MPI installation compatible with the host's MPI. This compatibility refers to ABI compatibility [37], as discussed in section 1.2.1.2. Similarly, ensuring compatibility between the runtime libraries required for GPU offloading is crucial; compatibility matrices for CUDA and ROCm are available at [24] and [1], respectively.

Once a suitable bootstrap image is identified, a Singularity `sandbox` is created, enabling interactive testing of Spack installations and model compilation. After successfully completing these tasks, the commands used in the sandbox can be transferred into a Singularity definition file for reproducibility.

The Spack installation is carried out within a Spack environment defined by a `spack.yaml` file, whose general structure was described in Section 1.2.2. In deploying all the necessary dependencies, we aimed at replicating the software stack of the host as closely as possible. This is not for compatibility—since the only libraries dynamically linked from the host to the container are those related to MPI and CUDA/ROCm—but rather to ensure a fair comparison of model performance between the container and the host.

For the compilation of the IFS model, an environment configuration file (`env.sh`) is typically created and tailored to the host's software stack. This file primarily requires specifying the installation paths of the packages needed by the model. In our design, we link these paths to the packages installed via Spack, as well as to the MPI/CUDA installations native to the bootstrap image.

2.2.1 Containerization on x86 architectures

The IFS model is mainly written in Fortran. For x86 architectures, it is primarily compiled with the `nvfortran` compiler from the NVIDIA NVHPC suite. The NVHPC suite [25] is a collection of compilers, libraries, and tools designed to support high-performance computing applications. It includes the `nvfortran`, `nvc`, and `nvc++` compilers, OpenMPI for distributed memory parallelism, and CUDA libraries for GPU acceleration.

As none of the target x86 architectures offer the `fakeroot` feature for container build, the builds had to be performed in a an external server, and the resulting SIF transferred afterwards to the specific clusters. For this purpose we employed a Virtual Machine endowed with `Ubuntu@22.04` hosted in the CINECA `ADA cloud` infrastructure [2].

Bootstrap image: The NVIDIA NVHPC SDK Catalog The NVIDIA HPC SDK catalog [27] is an open-access resource that offers base images preloaded with the NVHPC suite, tagged by compiler version, CUDA toolkit version, and operating system distribution.

The MPI shipped with the NVHPC suite is the `NVIDIA hpcx MPI` implementation [26], which is based on `OpenMPI v4.1.x`. Since OpenMPI guarantees ABI compatibility across all its versions [30], and the `nv` compilers are natively installed in the images from this catalog, these images are an ideal choice as bootstrap for x86 architectures. Regarding CUDA compatibility, it is necessary to choose an image with a CUDA installation compatible with the host system, see also the compatibility matrix [24].

In addition to offering a wide range of versioned base images, the catalog provides an additional feature. Most images are available in two variants: a `devel` variant, which is larger in size and includes all the OpenMPI, CUDA, and NVHPC compiler libraries necessary for compilation; and a `runtime` variant, which is more lightweight and contains only the components required for runtime execution. By leveraging a Singularity multistage build [33], see also Section 1.2.1.1, one can create a lightweight image optimized for fast shipment and deployment, using the `devel` variant for the first `Stage` of compilation and build, and a `runtime` one for the second stage, retaining the strictly necessary binaries and libraries only. For instance, the `23.1-devel-cuda12.0-ubuntu22.04` image weighs 4.97GB, while the corresponding `23.1-runtime-cuda12.0-ubuntu22.04` image weighs just 1.54GB.

For the containerization of our software stack we used the *same* base image from the HPC SDK catalogue both for ecWAM and IFS, namely:

- LEONARDO, MELUXINA: `23.1-devel-cuda_multi-ubuntu22.04`
- GALILEO100: `22.7-devel-cuda11.7-ubuntu22.04`

The base image employed for LEONARDO and MELUXINA is a `cuda_multi` distribution, holding three different flavors of CUDA, `11.0`, `11.8` and `12.0`: this allowed us to test and

compile the model with different version. For deployment we then switched to a runtime version, specifically `23.1-runtime-cuda11.8-ubuntu22.04`. It is worth noticing that the same image run without errors both on `Red Hat` distributions, LEONARDO, as well as on `Rocky` Linux ones, MELUXINA.

For GALILEO100, we had to use a slightly older base image, as the same one employed for the other two clusters would give runtime errors; we could not pinpoint the exact cause of this errors. In this case as well, differences in the OS between the host and container (resp, `CentOS` and `Ubuntu`) did not infer the deployment of the containerized model.

The Spack environment As mentioned earlier in the introduction to this section, the installation of the required software stack for the models within the container is carried out using the Spack environments feature.

In particular, once a Spack environment `spack_env` is created, details in Section 1.2.2, the corresponding `spack.yaml` file is populated with the compilers and software already installed in the base image via the commands:

```
spack compiler find
spack find
```

In particular, nvhpc, OpenMPI and CUDA are detected and used as dependencies for the additional software stack to be installed, which is specified in the `specs` section. Below, e.g., is a snapshot from the `spack.yaml` file used for the IFS container:

```
spack:
  config:
    source_cache: /spack_env/source_cache
    install_tree:
      root: /spack_env/install
    build_stage:
    - /spack_env/build/stage
    - /spack_env/spack-stage

  compilers:
  - compiler:
      spec: gcc@=11.4.0
      paths:
        cc:  /usr/bin/gcc
        cxx: /usr/bin/g++
        f77: /usr/bin/gfortran
        fc:  /usr/bin/gfortran
      operating_system: ubuntu22.04
      target: x86_64
  - compiler:
      spec: nvhpc@=23.1
      paths:
        cc:  /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/nvc
```

```
      cxx: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/nvc++
      f77: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/nvfortran
      fc:  /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/compilers/bin/nvfortran
    operating_system: ubuntu22.04
    target: x86_64

  specs:
  - hdf5@1.12.2            %nvhpc  -cxx +fortran +mpi +shared -szip
                                    build_system=cmake
  - zlib-ng                %gcc
  - fftw@3.3.10            %nvhpc  +mpi +openmp build_system=autotools
                                    precision=double,float
  - eigen@3.4.0            %nvhpc
  - boost@1.80.0           %nvhpc  +date_time +mpi +multithreaded
                                    +shared +filesystem
  - libaec@1.0.4           %nvhpc
  - netcdf-c@4.9.0         %nvhpc  +mpi +optimize +parallel-netcdf +shared
                                    +zstd build_system=autotools
  - netcdf-fortran@4.6.0   %nvhpc  +pic+shared build_system=autotools
  - parallel-netcdf@1.12.3 %nvhpc  +cxx+fortran+shared build_system=autotools
  - zstd@1.5.5 programs=true compression=lz4,lzma,zlib
  - cmake@3.24

  packages:
    all:
      compiler: [gcc@11.3.0, nvhpc@23.1]
      providers:
        mpi: [openmpi]
      variants: +gpfs +slurm +cuda cuda_arch=70 +nvml

    openmpi:
      externals:
      - spec: openmpi@4.1.5 +cuda +pmi fabrics=ucx schedulers=slurm
        prefix: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/comm_libs/hpcx/latest/ompi
      buildable: false
    cuda:
      externals:
      - spec: cuda@12.0.76
        prefix: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/cuda
      - spec: cuda@11.8.89
        prefix: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/cuda/11.8
      - spec: cuda@11.0.221
        prefix: /opt/nvidia/hpc_sdk/Linux_x86_64/23.1/cuda/11.0
    python:
      externals:
      - spec: python@3.10.12+bz2+readline+sqlite3+zlib
        prefix: /usr
```

```
      buildable: false
    [...]
```

Listing 2.1: spack.yaml for the RAPS48r1 container on x86

In the `config` section we define the path where the packages will be installed in the container, the `compiler` one is populated with the compilers provided with the bootstrap image; same goes for the `packages` section, which here we shortened leaving the most relevant packages. In the `specs` section we install the packages with variants as close as possible to those found in the host machine (LEONARDO), to provide a fair comparison between the host and container builds.

The Singularity definition file Once ensured that the necessary software stack would correctly install via Spack, we created a new `env.sh` file for RAPS, tailored to the specific container, listing the required library paths for compiling the model. Hence we built the model, always on a `sandbox` interactive environment, checking that the compilation phase would run without errors. Finally, we repacked the RAPS bundle with the updated `env.sh` file and reported the necessary commands for the container build in a Singularity definition file, (introduced in Section 1.2.1.1).

A completely analogue procedure was used for the containerization of ecWAM. Below, e.g., is a snapshot of the definition file used for the ecWAM container on LEONARDO and MELUXINA, compiled with `nvhpc@23.1` and provided with `CUDA@11.8` support:

```
Bootstrap: docker
From: nvcr.io/nvidia/nvhpc:23.1-devel-cuda_multi-ubuntu22.04
Stage: build

%files
    # copying spack.yaml file in the environment folder
    files/spack_recipes/spack_env-23.1.yaml    /spack_env/spack.yaml

    # copying ecwam bundle
    files/ecwam_bundle/*                       /ecwam_bundle/

    # copying scripts for build
    files/scripts_for_build-23.1/*             /scripts/

%post
    # Updating and installing spack dependencies
    apt-get update
    apt install -y wget apt-utils build-essential ca-certificates coreutils \
        curl unzip zip libssl-dev tree environment-modules gfortran git

    # Installing python packages needed for ecWAM
    pip install setuptools cython pywheel pyyaml fypp

    # Downloading and activating Spack
```

```
    cd /opt
    git clone --depth=2 https://github.com/spack/spack.git
    . /opt/spack/share/spack/setup-env.sh

    # Activating the Spack environment and installing
    cd /spack_env
    spack env activate -d .
    spack concretize --force
    spack install -y

    # Build for pure MPI
    echo "Compiling for MPI only.."
    . /scripts/MPI_build.sh

    # Build for GPU
    echo "Compiling for Cuda 11.8.."
    . /scripts/openACC_build_11.8.sh

    # Taking care of (to be) broken symlinks for runtime stage
    rm /usr/lib/x86_64-linux-gnu/libibverbs/libmlx5-rdmav34.so
    cp /usr/lib/x86_64-linux-gnu/libmlx5.so.1.23.40.0 \
        /usr/lib/x86_64-linux-gnu/libibverbs/libmlx5-rdmav34.so


##############################################################################

Bootstrap: docker
From: nvcr.io/nvidia/nvhpc:23.1-runtime-cuda11.8-ubuntu22.04
Stage: runtime

%files from build
    /spack_env/*                                    /spack_env/
    /opt/ecwam/install/*                            /opt/ecwam/install/
    /opt/ecwam_gpu_11.8/install/*                   /opt/ecwam_gpu/install/
    /lib/x86_64-linux-gnu/*                         /lib/x86_64-linux-gnu/
    /opt/spack_pkg                                  /opt/spack_pkg

%post
    # Removing unnecessary spack files for runtime
    cd /spack_env/
    rm -rf build/ mirror/ recipes/ source_cache/

    # Reinstalling
    apt-get install ucx-utils libucx-dev -y
```

Listing 2.2: Singularity Definition File for the ecWAM container on x86

In the `%files` section, we copy the `spack.yaml` file configured for `nvhpc@23.1`, along with

the ecWAM bundle and scripts required for the build process. Within the `%post` section, we install Spack's necessary dependencies using `apt`, along with specific `python` packages needed for the model. Next, we set up the required software in a Spack environment and proceed to compile the model. Subsequently, we transition to a lightweight version of the bootstrap image using a multistage build. In the `%files from build` section, we transfer necessary libraries and binaries from the previous stage. Finally, in the concluding `%post` section, we perform additional cleanup and reinstall the `ucx` libraries, which are inconveniently absent in the `runtime` images provided by the HPC SDK catalog.

2.2.2 Containerization on Cray: LUMI

For Cray architectures, the operational IFS is compiled using the `cee` compiler *(Cray Compiling Environment)*, which is part of the `HPE CPE` *(Cray Programming Environment)* suite. Similarly to the NVHPC suite for `x86`, CPE provides the necessary compilers and libraries for `Cray` architectures. This suite includes Cray compilers and libraries for both MPI and hardware acceleration on AMD GPUs, such as `cce`, `cray-mpich`, and `rocm`. Additionally, several other widely used HPC packages tailored for the Cray environment are provided, including `cray-libsci`, `cray-fftw`, `cray-python`, `cray-hdf5`, and `cray-netcdf`.

Base Image LUMI provides a `singularity-bindings` module, which allows simple base images with either `mpich` or `openmpi` to run with optimal performance on the cluster. However, to run and compile GPU-accelerated code for Cray-AMD systems, a base image with CPE is mandatory. This is primarily due to the need for specific AMD ROCm drivers and libraries, which are integral for GPU.

While HPE provides a catalog of container images with CPE at [15], access to these images is restricted by a licensing agreement. For this project, we used a base image provided by the CSC staff, which includes HPE CPE version `24.07`. This image is endowed with `cce@17.0.1` and `cce@18.0.1` Cray compilers, together with `cray-mpich/8.1.30`, and `rocm@6.0.3`.

It would be valuable to develop a similar workflow starting with a CPE container image that is either openly accessible or available through a licensed access.

The Spack Environment As much of the required software stack is already provided by CPE, we only had to install a few additional packages needed by IFS, and list the already available ones in the corresponding `packages` section. Below is a snapshot of the `spack.yaml` file used for the containerization of IFS on LUMI:

```
spack:
  config:
    source_cache: /spack_env/source_cache
    install_tree:
      root: /spack_env/install
    build_stage:
    - /spack_env/build/stage
    - /spack_env/spack-stage
```

```
9
10  specs:
11  - eigen@3.4.0   %cce
12  - boost@1.83.0  %cce
13  - cmake@3.27    %gcc
14  - libaec@1.0.6  %cce
15
16  compilers:
17  - compiler:
18      spec: cce@=18.0.0
19      paths:
20        cc:  /opt/cray/pe/cce/18.0.0/bin/craycc
21        cxx: /opt/cray/pe/cce/18.0.0/bin/crayCC
22        f77: /opt/cray/pe/cce/18.0.0/bin/crayftn
23        fc:  /opt/cray/pe/cce/18.0.0/bin/crayftn
24      operating_system: sles15
25      target: x86_64
26
27  - compiler:
28      spec: gcc@=13.3.0
29      paths:
30        cc:  /usr/bin/gcc-13
31        cxx: /usr/bin/g++-13
32        f77: /usr/bin/gfortran-13
33        fc:  /usr/bin/gfortran-13
34      operating_system: sles15
35      target: x86_64
36
37  packages:
38    mpi:
39      externals:
40      - spec: cray-mpich@8.1.30
41        prefix: /opt/cray/pe/mpich/8.1.30/ofi/crayclang/17.0/
42      buildable: false
43    cray-libsci:
44      externals:
45      - spec: cray-libsci@24.01.0
46        prefix: /opt/cray/pe/libsci/24.07.0/CRAYCLANG/17.0/x86_64
47      buildable: false
48    fftw:
49      externals:
50      - spec: fftw@3.3.10.8
51        prefix: /opt/cray/pe/fftw/3.3.10.8/x86_milan
52      buildable: false
53    python:
54      externals:
55      - spec: python@3.11.7
```

```
56        prefix: /opt/cray/pe/python/3.11.7
57      - spec: python@3.6.15
58        prefix: /usr
59    [...]
```

Listing 2.3: spack.yaml for the RAPS container on LUMI

The `spack.yaml` file in Listing 2.3 specifies the required `compilers`, including the Cray compiler (`cce@18.0.0`) and GCC compiler (`gcc@13.3.0`), as well as the `packages` provided by the CPE suite. In the `spec` section, we list the additional libraries required for RAPS that are not already included in the CPE suite (compare with the much larger `spec` section in Listing 2.1).

Singularity Definition File Following the same procedure as for the x86 containers, we created an ad hoc `env.sh` file to set up the necessary environment for the build process. The Singularity definition file below outlines the steps to build the RAPS container for LUMI. The `env.sh` file is packed into the image during the build phase and used to configure the environment during the containerization of the software. Following the same procedure used for the x86 containers, we created a custom `env.sh` file to configure the environment for the model build and repacked it in the `raps21_container.tar.gz` tar file. The Singularity definition file below outlines the steps to build the RAPS container for LUMI.

```
1  Bootstrap: localimage
2  From: ccpe-24.07-rocm-6.0_cce_18.0.1_v2.sif
3
4  %files
5      files/spack.yaml                                    /spack_env/
6      files/raps21_container.tar.gz                       /
7
8  %post
9
10     # Clone Spack repository
11     git clone --depth=2 https://github.com/spack/spack.git
12     . /spack/share/spack/setup-env.sh
13
14     # Set up and activate the Spack environment
15     cd /spack_env
16     spack env activate -pd .
17     spack concretize -f
18     spack install -y
19
20     # Unpack the provided tar file
21     cd / && tar -xvzf raps21_container.tar.gz
22
23     # Build RAPS with specified options
24     cd /raps && ./raps-build --with-single-precision --arch \
25         arch/eurohpc/lumi-c/cray-container/18.0.0/cray-mpich/8.1.30/env.sh \
```

```
26          --target ifsMASTER.SP -j 128 --build-dir=Lumi_cce18_build
27
28      # Run configuration and build scripts
29      cd Lumi_cce18_build/
30      ./configure.sh
31      ./build.sh
32
33  ### Reducing the size of the final image
34
35      # Remove tar and source files from the RAPS bundle
36      rm -rf /raps/Lumi_cce18_build/ifs_sp
37      rm -rf /raps/source
38      rm /raps21_container.tar.gz
39
40      # Remove the unused rocm
41      rm -rf /opt/rocm-6.0.3
42
43      # Remove the compilers except for the libraries, needed at runtime
44      # Temporarily move the 'lib' folders and delete unnecessary directories
45      mv /opt/cray/pe/cce/18.0.0/cce/x86_64/lib           /temp_lib_cce
46      mv /opt/cray/pe/cce/18.0.0/cce-clang/x86_64/lib     /temp_lib_clang
47      rm -rf /opt/cray/pe/cce
48
49      # Recreate the necessary directories and restore the 'lib' folders
50      mkdir -p /opt/cray/pe/cce/18.0.0/cce/x86_64     \
51              /opt/cray/pe/cce/18.0.0/cce-clang/x86_64
52      mv /temp_lib_cce/lib        /opt/cray/pe/cce/18.0.0/cce/x86_64
53      mv /temp_lib_clang/lib      /opt/cray/pe/cce/18.0.0/cce-clang/x86_64
54      rm -rf /temp_lib_cce /temp_lib_clang
```

Listing 2.4: Singularity Definition File for the RAPS container on LUMI

The `ccpe-24.07-rocm-6.0_cce_18.0.1_v2.sif` bootstrap image is the one provided by CSC. Unlike the x86 case, we did not have a runtime image available for use in a multistage build. To reduce the final image size, we manually removed unnecessary files, resulting in a .sif file that weighs approximately 3.5 GB, reduced from the original 8.6 GB.

Chapter 3

Performance Analysis

In this chapter, we analyze the performance of our containerized models and provide the runtime configurations (environment, modules, and bindings) necessary to execute the container images on the target machines. The performance analysis focuses on comparing the strong scaling profiles of the models when executed from a container versus an equivalent host installation.

We analyze the performance of the containerized ecWAM model across all x86 target clusters: GALILEO100, LEONARDO, and MELUXINA. On LEONARDO, we also present benchmarks for the RAPS 48r1 model, which incorporates OpenMP+MPI parallelism and GPU support. Although the *same* RAPS image was successfully tested on MELUXINA, we could not perform extensive benchmarks due to the lack of input files on the machine. Finally, for LUMI, we evaluate the performance of RAPS 49r1, which features OpenMP+MPI but lacks GPU support.

The goal of this project was to establish a consistent workflow for software containerization in HPC environments and to demonstrate that containerized software can run without significant performance degradation. As illustrated in the following sections, this goal has been achieved, with no more than a few percentage points of difference in the worst-case scenario. Furthermore, in multiple cases, the containerized software even outperformed the host build. Several explanations for this phenomenon could be offered:

- **Differences in software stack**: Even though we tried to build a software stack as similar as possible to the host one in the container, the latter could still be using slightly more optimized libraries.
- **Reduced system-level interference**: Containerized applications, being isolated, might operate with fewer interruptions or resource contention, particularly metadata contention [16].
- **Different MPI configuration**: The host and container versions of MPI are essentially identical in all cases. Nonetheless, there could be differences in the particular parameters configuration file deployed, or improved memory access due to buffering systems in the container.

With respect to the last point, an easy way to test MPI performance in a multinode environment is provided by the OSU Micro-Benchmarks [29]. This test suite is designed to

measure latency and bandwidth in communications through a variety of MPI routines, such as `pt2pt-bibw` (point-to-point bidirectional bandwidth), `bcast`, `allgather`, and similar. In Figure 3.1 below, we report the results of the `pt2pt-bibw` test run either from the host or from an OSU instance built inside the container, on the LEONARDO cluster. The figure shows the percentage difference in bandwidth and latency for increasing message sizes, comparing the container to the host. Better performance corresponds to larger bandwidth and smaller latency.

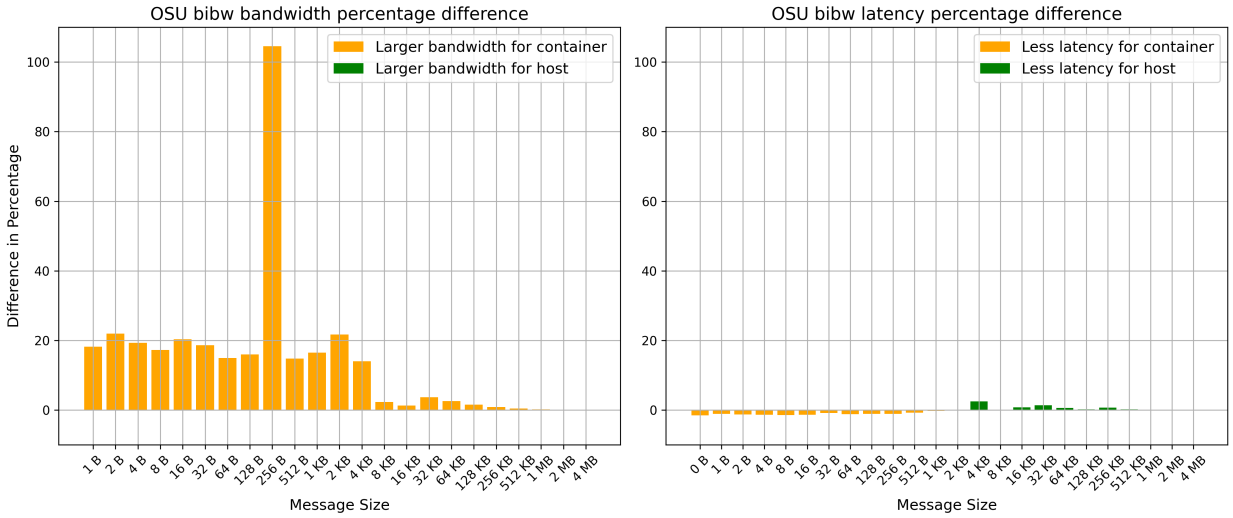

Figure 3.1: Performance comparison between containerized and host OSU instances. The left plot shows the percentage difference in bandwidth, and the right plot shows the percentage difference in latency. Larger bandwidth and smaller latency indicate better performance.

The latency, as measured by the OSU benchmarks, is practically identical in both cases. This indicates that there is *no overhead* introduced by the container in initializing MPI communication. On the other hand, the bandwidth measured when launching from the containerized OSU outperforms the host installation across all message sizes. The difference is about 20% up to `4KB` message sizes, with a peak of 100% at `256B`, and it gradually diminishes for larger message sizes. A possible explanation for this phenomenon could be related to the block size set in LEONARDO, which is exactly `4KB`: MPI communication is likely optimized to use the block size as the minimal size for data transfers while running on the host. The 100% peak at `256B` is more puzzling, but it could be due to particular configurations deployed by system administrators for the host OpenMPI installation, such as those specified in the `mca-params.conf` file.

3.1 Benchmarks on GALILEO100

In order to run the containerized ecWAM model, we first load the Singularity module

```
module load singularity/3.8.0--bind--openmpi--4.1.1
```

which in turn loads the `openmpi/4.1.1` module, which will provide the libraries necessary for a Singularity hybrid model execution, see Section 1.2.1.2.

The `singularity/3.8.0--bind--openmpi--4.1.1` module also defines a set of bindings necessary for container images to operate on GALILEO100:

```
setenv SINGULARITY_BIND /scratch_local,/g100_scratch,
/etc/passwd,/opt/slurm,/var/spool/slurmd/conf-cache,
/usr/lib64/libmunge.so.2,/usr/lib64/pmix/lib,/usr/lib64/pmix/mca_bfrops_v20.so,
/usr/lib64/pmix/mca_gds_hash.so,/usr/lib64/pmix/mca_psec_none.so,
/usr/lib64/pmix/mca_psensor_heartbeat.so,/usr/lib64/pmix/mca_ptl_tcp.so,
/usr/lib64/pmix/mca_bfrops_v12.so,/usr/lib64/pmix/mca_bfrops_v21.so,
/usr/lib64/pmix/mca_psec_munge.so,/usr/lib64/pmix/mca_gds_ds21.so,
/usr/lib64/pmix/mca_preg_native.so,/usr/lib64/pmix/mca_psec_native.so,
/usr/lib64/pmix/mca_psensor_file.so,/usr/lib64/pmix/mca_pshmem_mmap.so,
/usr/lib64/pmix/mca_ptl_usock.so,/lib64/libpmix.so.2,
/g100,/g100_work,/usr/lib64/liblustreapi.so.1,/usr/lib64/libkeyutils.so.1,
/usr/share/pmix/help-pmix-runtime.txt,/usr/lib64/libmca_common_dstore.so.1
```

These bindings ensure proper functionality of the containerized application by:

- Binding basic host filesystem paths for runtime writing (`/scratch_local`, `/g100_scratch`, `/g100_work`),
- Providing auxiliary paths required for SLURM execution, such as `/etc/passwd`, `/opt/slurm` and `/var/spool/slurmd/conf-cache`,
- Including the `munge` and `pmix` libraries necessary for running OpenMPI applications via `srun`. In particular, binding of `pmix` libraries are necessary in GALILEO100, as the native OpenMPI installation is configured to use `pmi2` rather than `pmix`.

Specific directories required by the model for runtime writing were also included,

```
RUNDIRS_BIND=$ECWAM_CACHE:/cache_dir,$ECWAM_DATA:/data_dir,$ECWAM_RUN:/run_dir
SINGULARITY_BIND=$RUNDIRS_BIND,$SINGULARITY_BIND
```

With this configuration, we conducted benchmarks on an increasing number of full nodes (from 4 to 32) of the `g100_usr_prod` partition of GALILEO100; for hardware specifics, refer to Section 1.3.1. The precise grid and dataset used for these tests (`etopo1_oper_an_fc_0640`) are detailed in the `ecWAM` repository [12]. For each fixed number of nodes, we launched 10 runs, from which we computed the average wallclock time. The command used to launch the executable is as follows:

```
mpirun -np $nodes singularity exec --env PREPEND_PATH=/opt/ecwam/bin \
    $WORK/ecwam.sif /opt/ecwam/bin/ecwam-chief
```

The results are shown in Figure 3.2 below.

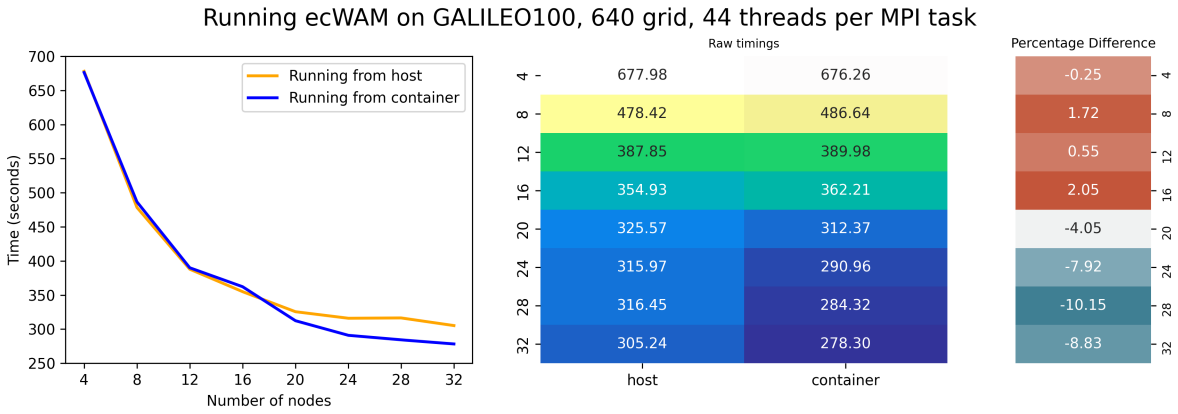

Figure 3.2: Strong scaling profiles for the ecWAM model run either from a container and from host on the GALILEO100 cluster.

In the figure we report a graphical plot of the strong scaling profiles, as well as raw timings and percentage differences. As revealed by the last column, the model run from the container has comparable performances with respect to the host, even outperforming the latter on large number of nodes. See the introduction to this Chapter 3 for possible explanations.

3.2 Benchmarks on LEONARDO

On LEONARDO, `SingularityPRO 3.11-5` is installed on the system, and no additional module needs to be loaded. The OpenMPI installation is already built with `pmix` support, so the necessary bindings include only the SLURM files and base filesystem paths for runtime writing:

```
export SINGULARITY_BIND=/leonardo_work,/leonardo_scratch,
/etc/passwd,/opt/SLURM,/var/spool/slurmd/conf-cache
```

On LEONARDO, we executed both the ecWAM and RAPS container images using the MPI-Singularity hybrid model with GPU support. The required module to be loaded (at the time of testing) is

```
module load openmpi/4.1.4--nvhpc--23.1-cuda-11.8
```

This module automatically loads `openmpi/4.1.4` and `Cuda/11.8`. The command used for execution is identical to the one employed on GALILEO100, with the addition of the `--nv` flag to enable GPU support:

```
mpirun -np $nodes singularity exec --nv --env PREPEND_PATH=/opt/ecwam/bin \
    $WORK/ecwam.sif /opt/ecwam/bin/ecwam-chief
```

For the ecWAM model, we tested the largest available `1280 grid` (see [12]) on an increasing number of full nodes (8 up to 256) of the LEONARDO `booster` partition; see Section 1.3.2. On each node, we ran with 4 MPI tasks, 8 CPUs per task, and 1 GPU per task. The results are shown in Figure 3.3. The performance results from both host and container executions are virtually identical.

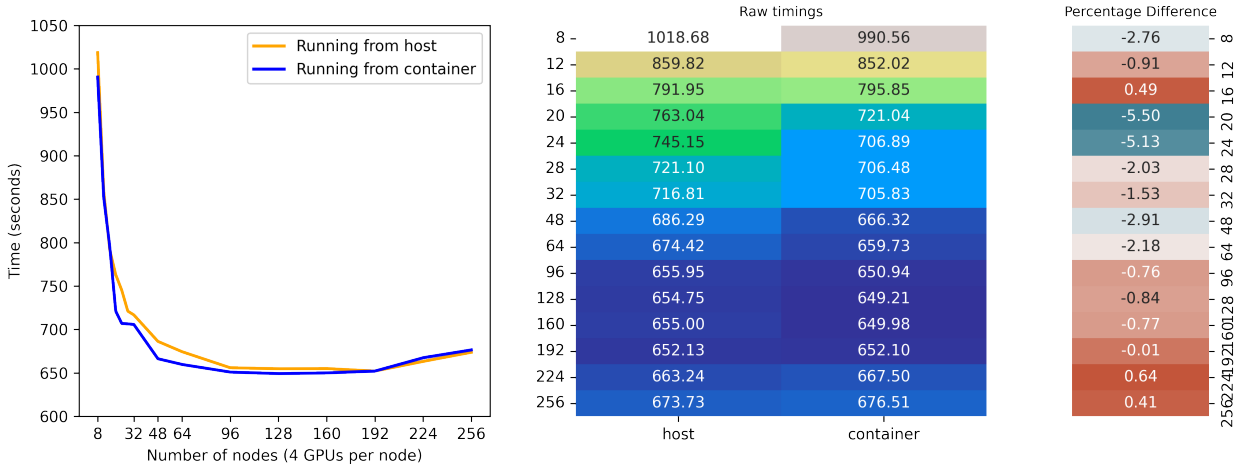

Figure 3.3: Comparison of host and container performance for the ecWAM model on the `1280 grid` in LEONARDO.

Similarly, for the IFS model, we tested the highest provided resolution, `tco2559l137` coupled withthe `NEMO` ocean model at a resolution of 1/4 of degree (see [28]). The setup is similar to the one employed for ecWAM: 4 MPI tasks per node, 8 CPUs per task, and 1 GPU per task, using 32 up to 256 nodes of the LEONARDO `booster` partition. The results are presented in Figure 3.4.

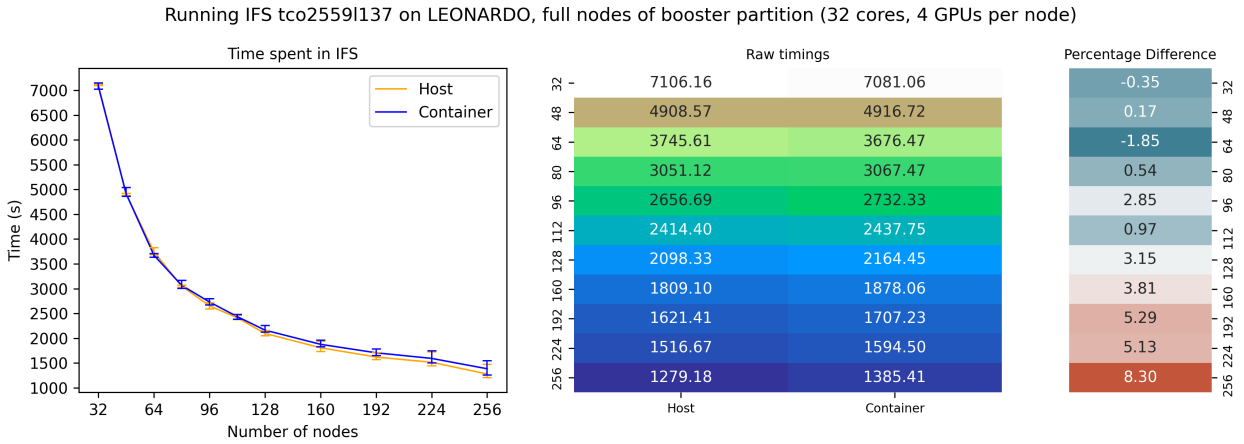

Figure 3.4: Performance comparison for the IFS model (`tco2559l137`) between host and container executions on LEONARDO.

The performance of the host and the container are comparable up to 128 nodes, but the container performance is slightly worse for configurations using a larger number of nodes.

A more detailed comparison for the 256-node configuration is shown in Figure 3.5, which reports the timings spent in each section of the model.

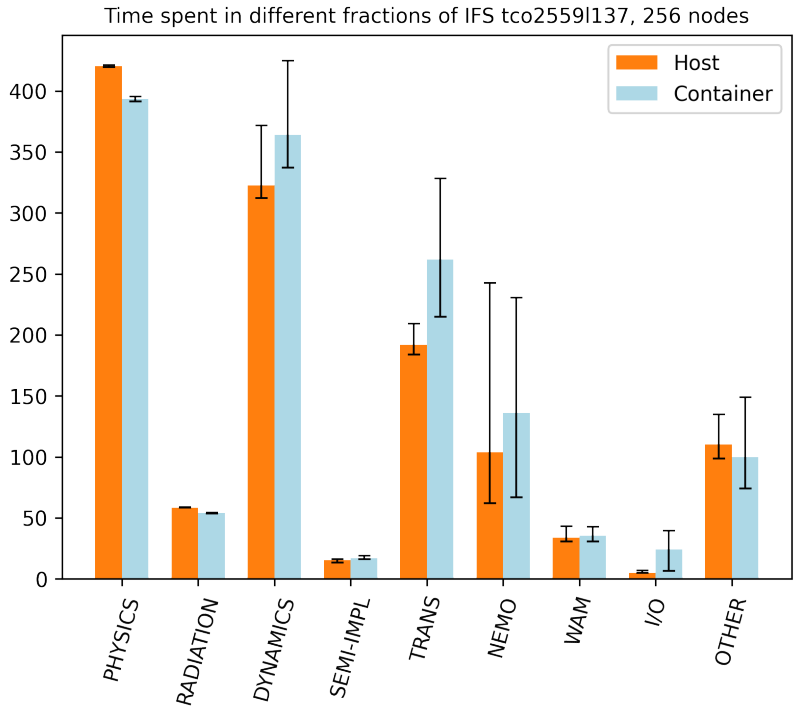

Figure 3.5: Section-wise timing comparison for the IFS model (`tco25591137`) on 256 nodes.

The container performs slightly worse in the `dynamics`, `trans`, and `nemo` sections, while outperforming the host in the `physics` section. The reasons for these discrepancies are not entirely clear but could potentially stem from subtle differences in the software stack between the two builds.

3.3 Benchmarks on MELUXINA

On MELUXINA, the `Apptainer` software is available rather than Singularity. The following modules were loaded for OpenMPI and CUDA:

```
module load Apptainer/1.2.4-GCCcore-12.3.0
module load OpenMPI/4.1.5-GCC-12.3.0
module load NVHPC/23.7-CUDA-11.7.0
```

The binded paths include only those related to SLURM and runtime writing. It is worth stressing that the *exact same images deployed on LEONARDO* for ecWAM and RAPS were successfully transferred and executed on MELUXINA. The only additional configurations required were the setting of two environment variables, identified with the help of MELUXINA user support:

- `export OMPI_MCA_btl="self,smcuda"`: configures communication to use shared memory and CUDA for local data transfers.
- `export UCX_TLS="rc_x,mm,cuda_copy,cuda_ipc,gdr_copy"`: sets the UCX transport layers, prioritizing GPU-optimized communication protocols.

For the ecWAM model, we tested both the medium `640 grid` and the larger `1280 grid` on full nodes (up to 48) of the MELUXINA `gpu` partition, see Section 1.3.3 for hardware specifics. WE run with 4 MPI tasks, 32 CPUs per task, and 1 GPU per task on each node. The results are shown in Figure 3.6.

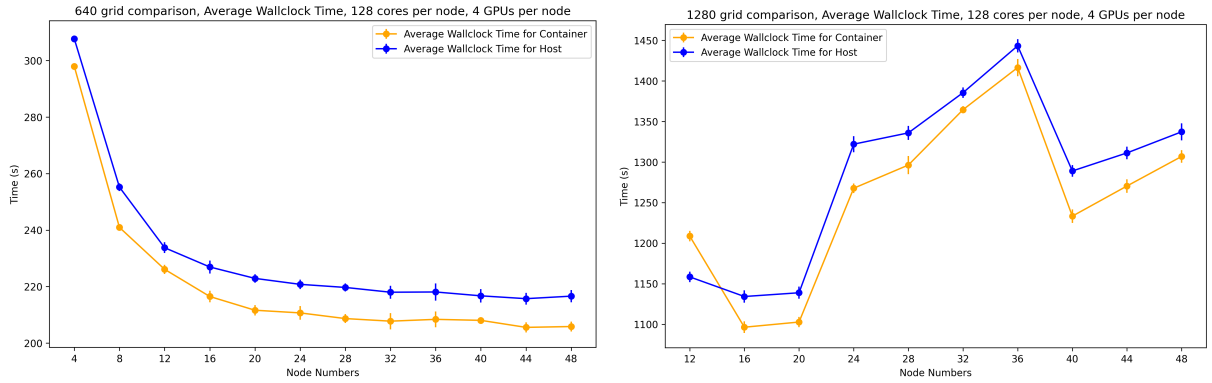

Figure 3.6: Performance comparison for the ecWAM model on the MELUXINA `gpu` partition. Left: `640 grid`; Right: `1280 grid`.

In both cases, the container slightly outperforms the host-based model. The strong scaling profile for the `640 grid` shows good scaling up to 16 nodes, flattening out for larger node counts. On the other side, the tests on the `1280 grid` exhibit poor scaling and alternate behavior across different node counts; nonetheless, the profiles of the two curves remain nearly identical, confirming no performance degradation passing from host to a container based version.

For the IFS model, we successfully tested the container image on MELUXINA using a coarse grid, namely `tco3991137`. However, the limited size of this grid does not permit a meaningful strong scaling analysis, and larger grids were not available on the machine.

3.4 Benchmarks on LUMI

On LUMI, the `singularity` software is installed system-wide, similarly to LEONARDO, so no module needs to be loaded. However, to enable applications launched from the container to utilize the cluster network, certain bindings are required:

```
export SINGULARITY_BIND=/opt/cray/libfabric,/usr/lib64/libcxi.so.1,
  /var/spool,/etc/host.conf,/etc/hosts,/etc/nsswitch.conf,
  /etc/resolv.conf,/etc/ssl/openssl.cnf
export SINGULARITYENV_LD_LIBRARY_PATH=/opt/cray/libfabric/1.15.2.0/lib64
```

In particular, `libfabric` and `libcxi` are bound from the host, as the versions inside the container are not properly configured to utilize the LUMI InfiniBand network. Additionally, the path to the `libfabric` libraries is appended to the container's `LD_LIBRARY_PATH`. The other bindings listed are required, as in previous cases, to interface with the SLURM scheduler. `libfabric` is a high-performance fabric software library that provides a unified API for accessing various network communication protocols; `libcxi`, on the other hand, is the Cray Communication Accelerator library, designed to enable efficient communication over Cray interconnects.

In order to run the container in a Singularity hybrid model fashion, the following MPI and network modules are loaded:

```
module load cray-mpich/8.1.29
module load craype-network-ofi
module load libfabric/1.15.2.0
```

On LUMI, we tested the RAPS 49r1 model on two different resolution grids: `tco1279l137` (medium resolution) and `tco2559l137` (large resolution). The benchmarks were conducted on the `LUMI-C` partition, see Section 1.3.4 for hardware specifics, utilizing up to 64 nodes, with 8 MPI tasks per node and 16 CPUs per task.

The strong scaling profiles for the `tco1279l137` grid are reported in Figure 3.7. In the plot, we compare both the overall wallclock times and the times spent in specific sections of the model: (`physics`, `dynamics`, and `trans`). The performances of the containerized and host-based models are comparable, with a slight advantage for the containerized execution.

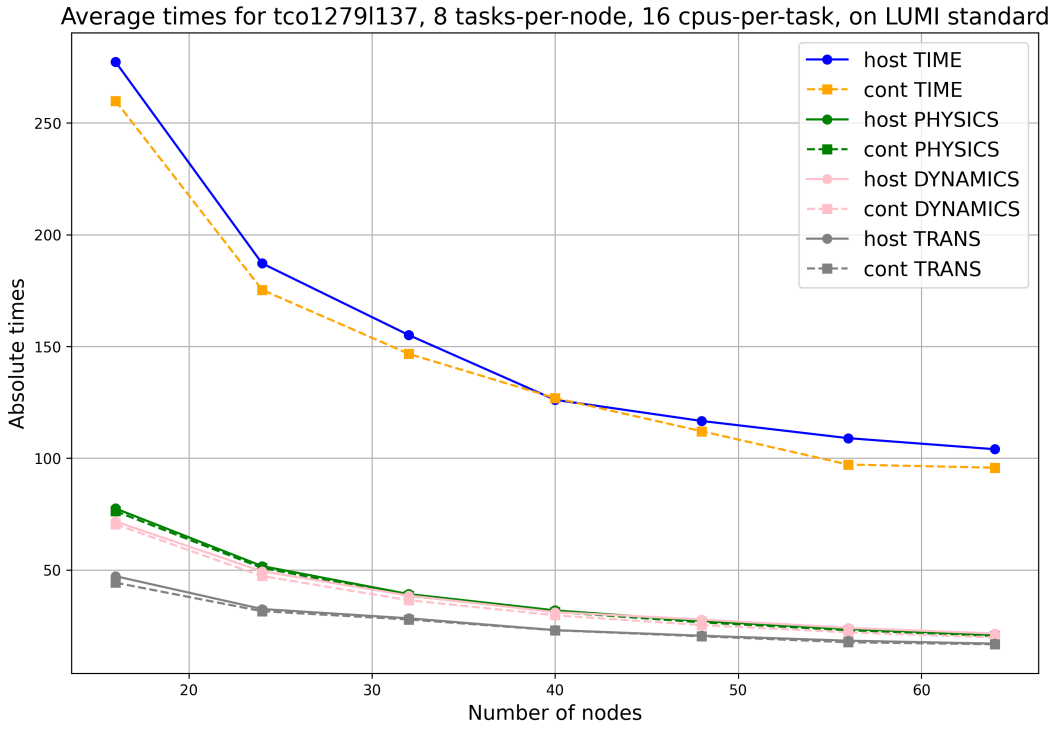

Figure 3.7: Strong scaling profile for the `tco1279l137` grid on LUMI, comparing wallclock times and specific sections of the model.

Similar results are observed for the larger `tco25591137` grid. A bar plot comparing the time spent in the different sections of the code when executed on 64 nodes of the `LUMI-C` partition is provided in Figure 3.8.

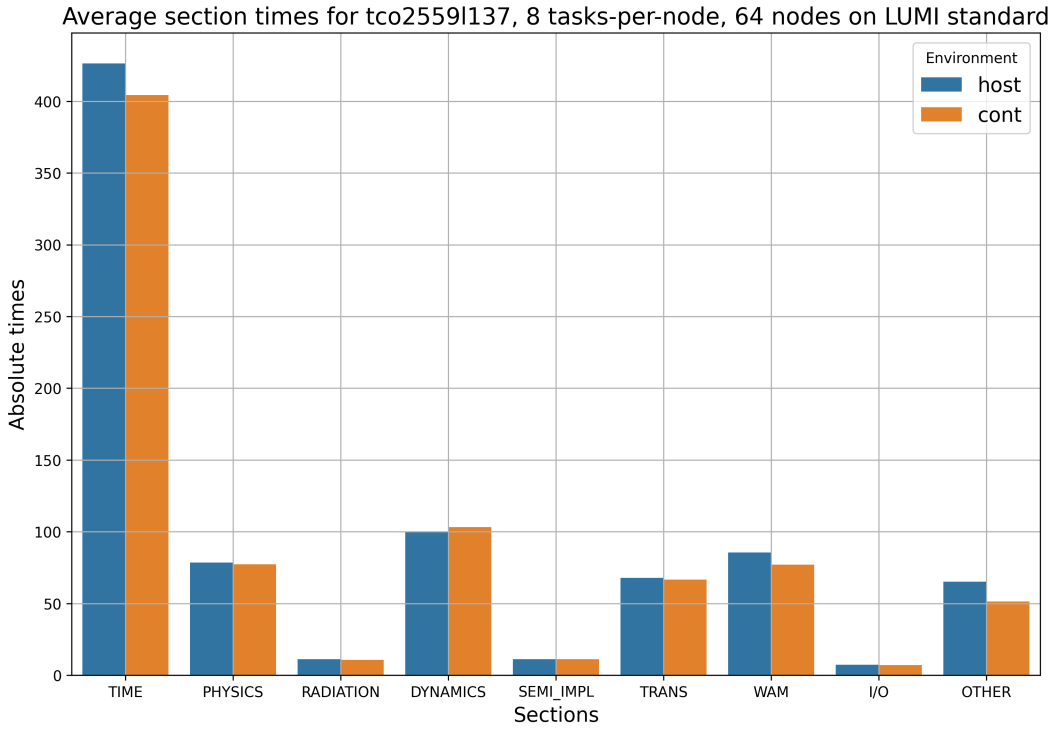

Figure 3.8: Comparison of time spent in different sections of the model for the `tco25591137` grid on 64 nodes of LUMI.

Chapter 4

Conclusions and Future Work

Achievements and Completed Tasks

This project has successfully established a robust and consistent workflow for containerizing weather models, namely the IFS and ecWAM model, across HPC clusters of varying architectures. The workflow revolves around two primary tools: **Singularity**, employed for container build and deployment, and **Spack**, used to manage the specific software stack requirements inside the containers. Together, these tools create a seamless yet rigorously defined workflow that balances flexibility with reproducibility.

A critical focus of the workflow has been on maintaining high performance and hardware compatibility for containerized applications. This was achieved by carefully defining the prerequisites for leveraging host HPC hardware effectively. Specifically:

- **MPI ABI compatibility** between the host and container MPI libraries ensures that the *Singularity hybrid model* can be used. The hybrid model is preferable to the alternative bind model, as it is significantly more portable and resilient to changes in the host software stack.
- **UCX ABI compatibility** between the host and container is required to utilize the high-performance InfiniBand network.
- **CUDA/ROCm compatibility** in terms of library and driver versions, enables GPU offloading, ensuring use of host accelerators.

The effectiveness of the workflow was validated through extensive performance testing. Applications featuring a hybrid level of parallelism (MPI + OpenMP) and GPU support were containerized and tested across multiple x86-based HPC clusters: GALILEO100, LEONARDO, and MELUXINA. Results showed negligible performance degradation compared to host-based builds, demonstrating the containers' ability to replicate and even enhance computational efficiency.

As part of the workflow, we leveraged the **NVIDIA HPC SDK catalog** to select base images for containerization on x86 architectures. These base images proved to be highly advantageous, providing a base software stack that includes: NVHPC compilers, an OpenMPI installation that is natively MPI and UCX ABI compatible with the target hosts,

and support for selectable CUDA versions. Moreover, they offer pre-built developer and runtime base images which facilitate multi-stage Singularity builds, resulting in lightweight yet functional final container images.

The workflow was also extended to Cray architectures, where we successfully built and deployed a containerized version of the IFS RAPS bundle on the LUMI supercomputer. The base image, which included an HPE CPE suite, was directly supplied by CSC User Support. The containerized model, which entailed MPI+OpenMP parallelism but did not include GPU offloading, was again shown to perform at a level comparable to a host-based installation.

Tasks in Progress and Future Work

While significant progress has been made, there remain key tasks to finalize and areas for further exploration.

First, we aim to complete the GPU execution of the IFS model on the LUMI cluster. Progress is currently hindered by the unavailability of a GPU-enabled release of RAPS, the software used to run IFS. The current RAPS version supports only CPU execution, and earlier versions are incompatible with LUMI's recent software updates.

Another challenge lies in standardizing the selection of base images for containerization on Cray architectures. For the LUMI deployment, we relied on a custom-crafted base image provided by CSC user support team. While this approach was successful, it lacks generalizability. An **HPE CPE container catalog** does exist, but its use is restricted by licensing constraints. Developing a systematic method or identifying freely accessible alternatives will be critical to scaling containerization efforts on Cray systems.

Finally, we are working to integrate the containerization workflow into a **CI/CD pipeline** to automate builds and deployments. The workflow here presented, based on Singularity definition files and Spack configuration files, is inherently suited for automation. However, several challenges must be addressed to achieve this integration effectively:

- **Dependency tracking**: HPC environments are dynamic, with frequent updates to software stacks and system configurations. A robust CI/CD pipeline must account for these changes.
- **Multi-architecture and multi-cluster constraints**: CI/CD tools must adapt to the diversity of systems, ensuring that builds are reproducible and deployable across architectures.
- **Performance validation**: Automation must include rigorous performance checks to confirm that containerized applications maintain parity with host-based versions.

Addressing these challenges will not only streamline the workflow but also enhance its scalability and utility for managing evolving weather models in HPC environments.

Bibliography

[1] AMD, *ROCm compatibility matrix*. https://rocm.docs.amd.com/en/latest/compatibility/compatibility-matrix.html.

[2] CINECA, *ADA Cloud user guide*. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.5%3A+ADA+Cloud+UserGuide.

[3] ——, *Cineca homepage*. https://www.cineca.it/it.

[4] ——, *GALILEO100 hardware overview*. https://www.hpc.cineca.it/systems/hardware/galileo100/.

[5] ——, *GALILEO100 user guide*. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.3%3A+GALILEO100+UserGuide.

[6] ——, *LEONARDO hardware overview*. https://leonardo-supercomputer.cineca.eu/it/leonardo-hpc-system/.

[7] ——, *LEONARDO user guide*. https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2%3A+LEONARDO+UserGuide.

[8] R. Cox, *Version SAT: Dependency hell is NP-complete. But maybe we can climb out.* Available at: https://research.swtch.com/version-sat, Dec 2016.

[9] Destination Earth Initiative, *DE homepage*. https://destination-earth.eu/.

[10] Docker Inc., *Docker homepage*. https://hub.docker.com/.

[11] EuroHPC Joint Undertaking. https://eurohpc-ju.europa.eu/.

[12] European Centre for Medium-Range Weather Forecasts (ECMWF), *ECMWF Ocean Wave Model (ecWAM)*. https://github.com/ecmwf-ifs/ecwam.

[13] ——, *Homepage*. https://www.ecmwf.int/.

[14] ——, *Integrated Forecasting System (IFS)*. https://www.ecmwf.int/en/forecasts/documentation-and-support/changes-ecmwf-model.

[15] Hewlett Packard Enterprise (HPE), *HPE support page*. https://support.hpe.com.

[16] D. M. Jacobsen and R. S. Canon, *Contain this, unleashing docker for hpc*, Proceedings of the Cray user Group, (2015), pp. 33–49. Available at: https://www.nersc.gov/assets/Uploads/cug2015udi.pdf.

[17] G. M. Kurtzer, V. Sochat, and M. W. Bauer, *Singularity: Scientific containers for mobility of compute*, PloS one, 12 (2017).

[18] LUMI Consortium, *LUMI hardware overview*. https://docs.lumi-supercomputer.eu/hardware/.

[19] ——, *LUMI user guide*. https://docs.lumi-supercomputer.eu/.

[20] LuxProvide, *MeluXina hardware overview*. https://docs.lxp.lu/system/overview/.

[21] ——, *MeluXina user guide*. https://docs.lxp.lu/.

[22] MHPC (Master in High Performance Computing), *Mhpc homepage*. https://www.mhpc.it/.

[23] NEMO Consortium, *Homepage*. https://www.nemo-ocean.eu/.

[24] NVIDIA Corporation, *CUDA compatibility matrix*. https://docs.nvidia.com/deploy/cuda-compatibility.

[25] ——, *NVIDIA HPC Software Development Kit (SDK)*. https://developer.nvidia.com/hpc-sdk.

[26] ——, *NVIDIA HPCX*. https://developer.nvidia.com/networking/hpc-x.

[27] ——, *NVIDIA NVHPC SDK container catalog*. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/nvhpc.

[28] OceanPredict, *Ocean models descriptions*. https://oceanpredict.org/science/operational-ocean-forecasting-systems/ocean-models/.

[29] Ohio State University, *OSU Micro-Benchmarks*. https://mvapich.cse.ohio-state.edu/benchmarks/.

[30] OpenMPI Developers, *OpenMPI ABI/API compatibility*. https://docs.open-mpi.org/en/v5.0.x/version-numbering.html#api-and-abi-compatibility.

[31] Spack Developers, *Spack homepage*. https://spack.io/.

[32] Sylabs, *Singularity MPI Hybrid Model*. https://docs.sylabs.io/guides/3.9/user-guide/mpi.html#hybrid-model.

[33] ——, *Singularity multistage build guide*. https://docs.sylabs.io/guides/3.9/user-guide/definition_files.html#multi-stage-builds.

[34] ——, *Singularity user guide*. https://docs.sylabs.io/guides/latest/user-guide/.

[35] ——, *Singularity/Apptainer homepage*. https://sylabs.io/.

[36] TOP500 Project, *Top 500 supercomputers*. https://top500.org/.

[37] UCX Team, *UCX ABI compatibility timeline*. https://abi-laboratory.pro/index.php?view=timeline&l=ucx.