Using Apptainer on the Cluster (replaces Singularity)



Apptainer will replace Singularity on the the LUH-Clusters. Currently you can use both commands `apptainer` and `singularity` because the last one is a symlink to `apptainer`. This may change in the future.

Please note: This instruction has been written for Apptainer 1.3.3-*

Please note: If you would like to fully manage your apptainer container images directly on the cluster, including build and/or modify actions, please contact us and ask for the permission “apptainer fakeroot” to be added to your account (because you will need it).

Apptainer enables users to execute containers on High-Performance Computing (HPC) cluster like they are native programs or scripts on a host computer. For example, if the cluster system is running CentOS Linux, but your application runs in Ubuntu, you can create an Ubuntu container image, install your application into that image, copy the image to an approved location on the cluster and run your application using Apptainer in its native Ubuntu environment.

The main advantage of Apptainer is that containers are executed as an unprivileged user on the cluster system and, besides the local storage `TMPDIR`, they can access the network storage systems like `HOME`, `BIGWORK` and `PROJECT`, as well as GPUs that the host machine is equipped with.

Additionally, Apptainer properly integrates with the Message Passing Interface (MPI), and utilizes communication fabrics such as InfiniBand and Intel Omni-Path.

If you want to create a container and set up an environment for your jobs, we recommend that you start by reading the Apptainer documentation. The basic steps to get started are described below.

Building Apptainer container using a recipe file

If you already have a pre-build container ready for use, you can simply upload the container image to the cluster and execute it. See the section below about running container images.

Below we will describe how to build a new or modify an existing container directly on the cluster. A container image can be created from scratch using a recipe file, or fetched from some remote container repository. In this sub-section, we will illustrate a recipe file method. In the next one, we will take a glance at remote container repositories.

Using a Apptainer recipe file is the recommended way to create containers if you want to build reproducible container images. This example recipe file builds a RockyLinux 9 container:

rocky9.def

```
BootStrap: yum
OSVersion: 9
MirrorURL: https://ftp.uni-
hannover.de/rocky/%{OSVERSION}/BaseOS/$basearch/os
Include: yum

%setup
  echo "This section runs on the host outside the container during
bootstrap"

%post
  echo "This section runs inside the container during bootstrap"

  # install packages in the container
  yum -y groupinstall "Development Tools"
  yum -y install wget vim python3 epel-release
  yum -y install python3-pip

  # install tensorflow
  pip3 install --upgrade tensorflow

  # enable access to BIGWORK and PROJECT storage on the cluster system
  mkdir -p /bigwork /project

%runscript
  echo "This is what happens when you run the container"

  echo "Arguments received: $*"
  exec /usr/bin/python3 "$@"

%test
  echo "This test will be run at the very end of the bootstrapping
process"

  /usr/bin/python3 --version
```

This recipe file uses the `yum` bootstrap module to bootstrap the core operation system, RockyLinux 9, within the container. For other bootstrap modules (e.g.. `docker`) and details on apptainer recipe files, refer to the online documentation.

The next step is to build a container image on one of the cluster login servers.

Note: your account must be authorized to use the `--fakeroot` option. Please contact us at cluster-help@luis.uni-hannover.de.

Note: Currently, the `--fakeroot` option is enabled only on the cluster login nodes.

```
username@login01$ apptainer build --fakeroot rocky9.sif rocky9.def
```

This creates an image file named `rocky9.sif`. By default, apptainer containers are built as read-only SIF(Apptainer Image Format) image files. Having a container in the form of a file makes it easier to transfer it to other locations both within the cluster and outside of it. Additionally, a SIF file can be signed and verified.

Note that a container as the SIF file can be built on any storage of the cluster you have a write access to. However, it is recommended to build containers either in your `$BIGWORK` or in some directory under `/tmp` (or use the variable `$MY_APPTAINER`) on the login nodes.

Note: Containers located only under the paths `$BIGWORK`, `$SOFTWARE` and `/tmp` are allowed to be executed using `shell`, `run` or `exec` commands, see the section below,

The latest version of the `apptainer` command can be used directly on any cluster node without prior activation.

Downloading containers from external repositories

Another easy way to obtain and use a Apptainer container is to retrieve pre-build images directly from external repositories. Popular repositories are Docker Hub or Apptainer Library. You can go there and search if they have a container that meets your needs. For docker images, use the search form at Docker Hub instead.

In the following example we will pull the latest python container from Docker Hub and save it in a file named `python_latest.sif`:

```
username@login01$ apptainer pull docker://python:latest
```

The `build` sub-command can also be used to download images, where you can additionally specify your preferred container file name:

```
username@login01$ apptainer build my-ubuntu22.04.sif
library://library/default/ubuntu:22.04
```

How to modify existing Apptainer images

First you should check if you really need to modify the container image. For example, if you are using Python in an image and simply need to add new packages via `pip` you can do that without modifying the image by running `pip` in the container with the `--user` option.

To modify an existing SIF container file, you need to first convert it to a writable sandbox format.

Please note: Since the `--fakeroot` option of the `shell` and `build` sub-commands does not work with container sandbox when the container is located on a shared storage such as `BIGWORK`, `PROJECT` or `HOME`, the container sandbox must be stored locally on the login nodes. We recommend using the `/tmp` directory (or variable `$MY_APPTAINER`) which has sufficient capacity.

```
username@login01$ cd $MY_APPTAINER
```

```
username@login01$ apptainer build --sandbox rocky9-sandbox rocky9.sif
```

The `build` command above creates a sandbox directory called *rocky9-sandbox* which you can then `shell` into in writable mode and modify the container as desired:

```
username@login01$ apptainer shell --writable --fakeroot rocky9-sandbox
Apptainer> yum install -qy python3-matplotlib
```

After making all desired changes, you exit the container and convert the sandbox back to the SIF file using:

```
Apptainer> exit
username@login01$ apptainer build -F --fakeroot rocky9.sif rocky9-sandbox
```

Note: you can try to remove the sandbox directory *rocky9-sandbox* afterward but there might be a few files you can not delete due to the namespace mappings that happens. The daily `/tmp` cleaner job will eventually clean it up.

Running container images

Please note: In order to run a Apptainer container, the container SIF file or sandbox directory must be located either in your `$BIGWORK`, in your group's `$SOFTWARE` or in the `/tmp` directory.

There are four ways to run a container under Apptainer.

If you simple call the container image as an executable or use the Apptainer `run` sub-command it will carry out instructions in the `%runscript` section of the container recipe file:

How to call the container SIF file:

```
 username@login01:~$ ./rocky9.sif --version
This is what happens when you run the container
Arguments received: --version
Python 3.8.6
```

Use the `run` sub-command:

```
 username@login01:~$ apptainer run rocky9.sif --version
This is what happens when you run the container
Arguments received: --version
Python 3.8.6
```

The Apptainer `exec` sub-command lets you execute an arbitrary command within your container instead of just the `%runscript`. For example, to get the content of file `/etc/os-release` inside the container:

```
 username@login01:~$ apptainer exec rocky9.sif cat /etc/os-release
NAME="Rocky Linux"
VERSION="8.4 (Green Obsidian)"
```

```
....
```

The Apptainer `shell` sub-command invokes an interactive shell within a container. Note the `Apptainer>` prompt within the shell in the example below:

```
 username@login01:$ apptainer shell rocky9.sif
Apptainer>
```

Note that all three sub-commands `shell`, `exec` and `run` let you execute a container directly from remote repository without first downloading it on the cluster. For example, to run an one-liner “Hello World” ruby program:

```
 username@login01:$ apptainer exec library://sylabs/examples/ruby ruby -e
'puts "Hello World!"'
Hello World!
```

Please note: You can access (read & write mode) your HOME, BIGWORK and PROJECT (only login nodes) storage from inside your container. In addition, the `/tmp` (or TMPDIR on compute nodes) directory of a host machine is automatically mounted in a container. Additional mounts can be specified using the `--bind` option of the `exec`, `run` and `shell` sub-commands, see `apptainer run --help`.

Apptainer & parallel MPI applications

In order to containerize your parallel MPI application and run it properly on the cluster system you have to provide MPI library stack inside your container. In addition, the userspace driver for Mellanox InfiniBand HCAs should be installed in the container to utilize cluster InfiniBand fabric as a MPI transport layer.

This example Apptainer recipe file `ubuntu-openmpi.def` retrieves an `Ubuntu` container from `Docker Hub`, and installs required MPI and InfiniBand packages:

Ubuntu 20.04

ubuntu-openmpi.def

```
BootStrap: docker
From: ubuntu:focal

%post
# install openmpi & infiniband
apt-get update
apt-get -y install openmpi-bin openmpi-common libibverbs1 libmlx4-1

# enable access to BIGWORK storage on the cluster
mkdir -p /bigwork /project
```

```
# enable access to /scratch dir. required by mpi jobs
mkdir -p /scratch
```

Ubuntu 22.x - 24.x

ubuntu-openmpi.def

```
BootStrap: docker
From: ubuntu:latest

%post
# install openmpi & infiniband
apt-get update
apt-get -y install openmpi-bin openmpi-common ibverbs-providers

# enable access to BIGWORK storage on the cluster
mkdir -p /bigwork /project

# enable access to /scratch dir. required by mpi jobs
mkdir -p /scratch
```

Once you have built the image file `ubuntu-openmpi.sif` as explained in the previous sections, your MPI application can be run as follows (assuming you have already reserved a number of cluster compute nodes):

```
module load GCC/10.2.0 OpenMPI/4.0.5
mpirun apptainer exec ubuntu-openmpi.sif /path/to/your/parallel-mpi-app
```

The above lines can be entered at the command line of an interactive session, or can also be inserted into a batch job script.

Writable overlay images for high-I/O workloads

If your application workflow requires creating, modifying, or reading many small files at runtime (such as installing Python packages via `pip`, caching Deep Learning models, or accessing uncompressed datasets), you should **not** store such file collections directly as ordinary directory structures on your `$BIGWORK`. This restriction also applies to uncompressed Apptainer sandbox directories.

Large numbers of individual files consume one inode per file and generate significant metadata operations on the Lustre-based `BIGWORK` filesystem. This can lead to the exhaustion of your `$BIGWORK` inode quota and an increased metadata load for the entire cluster.

Instead, you can combine a standard read-only `.sif` container image with a **single-file writable overlay image**. From the perspective of the Lustre filesystem, the overlay therefore consumes only

one inode, regardless of how many files are stored inside it. Within the container, the overlay provides a writable filesystem layer on top of the read-only SIF image.

This significantly reduces the number of Lustre filesystem objects and metadata operations compared with storing the same files directly as a directory tree.

Paths outside the overlay

Only files written to paths provided by the overlay benefit from this mechanism.

Paths that are automatically bind-mounted from the cluster host, such as `$HOME`, `$BIGWORK`, `$PROJECT`, and `/tmp`, are outside the container's overlay filesystem. Files written to these paths are written directly to the corresponding host filesystem.

Therefore, high-I/O workloads should use an appropriate internal container path that is not covered by a host bind mount, for example `/opt/my-software/`.

Do not assume that an arbitrary path inside the container is backed by the overlay. Verify the bind configuration of your Apptainer environment if necessary.

Creating a writable overlay image

You can initialize a sparse EXT3 filesystem image file directly on a login node. In this example, we create a 5 GB sparse overlay file named `my-overlay.img`:

```
username@login01$ cd $BIGWORK
username@login01$ apptainer overlay create --size 5120 --sparse my-
overlay.img
```

The `--size` argument is defined in MiB. The `--sparse` flag ensures the file only consumes actual disk space as data is written, up to the defined size limit.

The overlay size is a fixed filesystem limit. Once the overlay filesystem is full, further writes will fail even if sufficient space remains available in your `$BIGWORK`. Choose the overlay size according to the expected amount of data. An existing overlay cannot simply grow dynamically during container execution.

Preparing and using the overlay environment

You do not need elevated Apptainer privileges like `--fakeroot` to modify the container environment when using an overlay. Launch your container with the `--overlay` flag to unlock write capabilities across the internal system paths. You can then interactively create directories and install software:

```
username@login01$ apptainer shell --overlay my-overlay.img rocky9.sif
Apptainer> mkdir /opt/my-software
Apptainer> pip3 install --target=/opt/my-software scipy
Apptainer> exit
```

All directories and files created under `/opt/my-software` are automatically encapsulated within your single `my-overlay.img` file on the host Lustre filesystem.

Execution in SLURM batch scripts

When submitting production jobs to compute nodes, pass your configured overlay file using the `--overlay` flag. Ensure your application's input, output, or cache paths point strictly to your overlay-backed container directories rather than to host-mounted paths:

example_apptainer_overlay.sh

```
#!/bin/bash -l
#SBATCH --job-name=example_apptainer_overlay
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --mem-per-cpu=3G
#SBATCH --gres=gpu:1
#SBATCH --output my_apptainer-job_%j.out
#SBATCH --partition=gpus
#SBATCH --time=02:00:00

# Load modules
module load NVHPC/26.5-CUDA-13.2.0

# Change to work dir
cd $SLURM_SUBMIT_DIR

# Execute the application targeting your overlay-backed path
apptainer exec --overlay my-overlay.img rocky9.sif python3 train_llm.py
--output_dir /opt/my-software/outputs --cache_dir /opt/my-
software/cache
```

Files written to `/opt/my-software/outputs` and `/opt/my-software/cache` are stored inside the overlay `my-overlay.img`, provided these paths are not covered by host bind mounts.

Concurrent use of an overlay

A writable overlay image must not be used concurrently by multiple processes or jobs that modify its contents. An overlay image contains an ext3 filesystem and should be treated similarly to a filesystem mounted for writing. Concurrent writes from multiple containers or SLURM jobs can result in filesystem corruption. For parallel workflows, you must generate a separate overlay image file for each concurrent job task.

Further Reading

- Apptainer home page
- Apptainer Library
- Docker Hub

From:
https://docs.cluster.uni-hannover.de/ - **Cluster Docs**

Permanent link:
https://docs.cluster.uni-hannover.de/doku.php/guide/apptainer

Last update: **2026/09/15 17:03**