GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

[Building a reasonable sized Gromacs container image]

[Winfried Oed]

MatrNr: [21674445]

Supervisor: [Azat Khuziyakhmetov]

Georg-August-Universität Göttingen
Institute of Computer Science

March 29, 2023

Abstract

Containers are a great tool to distribute software into High-Performance Computing (HPC) systems. Building them and shrink their size can be however cumbersome. Installing the molecular dynamics simulation software *GROMACS* [Bek+] [1] inside a container requires many steps. Azat Khuziyakhmetov facilitated this process by implementing a container image building approach that uses multiple recipes and images. Specific versions can be defined in these recipes and by calling a shell build script the containers are subsequently build. Since compiling software needs additional library's the final container image is not optimized in size. During this course the build script was extended to streamline the building process. Software versions can now easily be defined in one place inside the build script and a few extra parameters like the builder to use are added. The container size can be minimized using two different methods and the process generates a lightweight container image. Finally the image can be used on the HPC system. All source code is available and further usage instructions can be found in the repository [2].

[1] https://www.gromacs.org/
[2] https://gitlab.com/Winnus/gromacs-container-for-hpc

Contents

List of Tables iii

List of Figures iii

List of Listings iii

List of Abbreviations iv

1 Introduction 1

2 Background 2

2.1 HPC Container Maker 2

3 Methodology 3

3.1 build script 4

3.2 Expected outcome 5

4 Implementation 5

4.1 Deploy methods 5

5 Evaluation/Results 6

6 Challenges / Discussion 7

7 Conclusion 8

References 9

A Appendix A1

B Figures A1

C Code samples A1

List of Tables

1 Sizes for the different container images generated during the build steps. Note that the software column includes the installation or removal of all needed dependencies -which are not listed-. The root image is not build, instead it is pulled from the *Docker* library at "*docker.io/library/ubuntu:22.04*". 6

List of Figures

1 Schematic drawing of specific layers in a container image. Whenever a new run statement is made a new layer is added to the image. *myfile* is added into the second layer of the image due to the first run command. The second run command removes the file and as such it is not present anymore in the third layer of the image. However it is still present in layer two of the image and as such contributes to the overall size of the image. . A1

List of Listings

1 *HPC Container Maker (HPCCM)* definition file which will install *Open Message Passing Interface (MPI)*. The resulting *Docker* container definition file can be seen in the appendix, listing 5. 3
2 Variable section of the build script containing all variables that can be modified. A2
3 HPCCM recipe to remove the compiler and software garbage with spack. A shell command is specified which: uses spack to uninstall the compiler from the system wide packages, doing a garbage collection on the system wide installed spack files, loads the spack environment in which *gromacs* is installed, doing a garbage collection inside that environment. A3
4 HPCCM recipe to remove the compiler and software garbage with spack and copy the cleaned *spack* environment into freshly build deploy image. *Spack* specific setups are done as well. A4
5 Docker definition file to install *openmpi* in a container. The definition file was automatically generated by *HPCCM* using the recipe listed in 1 . . . A5

List of Abbreviations

HPC High-Performance Computing

HPCCM HPC Container Maker

MPI Message Passing Interface

GCC GNU Compiler Collection

FFTW Fastest Fourier Transform in the West

1 Introduction

In multi user environments it is not possible for the administrators to provide all software the users might need. Especially different users might want different versions of the same software, leading to an unmanageable administration case. Software containers [Heo+18] [KSB17] provide a solution here because they can be build by the user and ideally contain all the necessary software within themselves. As such the system administrator has not to deal with different software stacks. He only needs to maintain the container runtime library's and some specific software that can not be shipped within a container such as message passing interfaces (MPI) like Open MPI [3]. These software can not be shipped inside a container because it needs to be installed and configured on the system wide nodes, a task the user can not perform due to insufficient rights and knowledge. Containers have also the benefit of being easily shared among users. Additionally they represent a certain state of a software stack and as such, if something went wrong, software can be easily resetted into a clean state. In comparison to virtual machines they are light weight for the host system. However containers have the drawback that they should not run with root permissions as any suspicious software would gain root access to the host system. Here VM's are better suited as the emulated hardware environment is separated to the host system. However software running in a VM can not be integrated on HPC nodes as well as containerized software. Hence rootless containers are the solution and are adapted in HPC software distribution.

Building containers requires the user to write container definition files and building the container on their own. To have an easy to use container build scrip for the molecular dynamic simulation software *Gromacs* [Bek+] Azat Khuziyakhmetov wrote a container build script. The script uses the HPCCM [4] to generate *Docker* definition files that and building containers from them. To compile and install -inside the containers- software that is hardware optimized the flexible software manager *Spack* [5] is used. As software compilation takes a lot of additional library's the final container image is not optimized in size. In this work the final image size was optimized implementing two solutions. Additionally the build script was adapted to facilitate the container build process even more. All necessary variables can easily be modified and adapted to ones needs in one place. In the end building a reasonable sized container image only containing the needed runtime library's for *Gromacs* is not a difficult task with the provided work. The resulting container image is roughly 51 percent smaller compared to the non optimized one, enabling efficient use on the HPC system.

Contribution of this work to the already existing solution.

1. shrink the image size of the final container image.
2. improve the overall usability of the container build process
3. provide a documented procedure of how to generate and use a *Gromacs* container image on the HPC system [6].

[3] `https://www.open-mpi.org/`
[4] https://github.com/NVIDIA/hpc-container-maker
[5] `https://github.com/spack/spack`
[6] `https://gitlab.com/Winnus/gromacs-container-for-hpc`

4. provide a detailed report of how different tools are combined to facilitate a container build process.

This report is structured as follows. In section 2 basic information about container definitions and used software are given. Section 3 will describe the container build approach implemented in this work in more detail and the expected outcome. Next, section 4 will describe some finer details about the implementation and used commands. In section 5 the achievements and results are discussed. The following section 6 will discuss problems that occurred and compare the implemented container build-flow with the *Spack* inbuild one. Finally the report ends with a conclusion in section 7.

2 Background

Containers need a specification which defines how they are structured and how to the container daemon should run them. *OCI* [7] is an open and widely used container standard in the container world. Most daemons like *Docker* or *Podman* work with it. As such containers compiled by different daemon can run on another. The build instructions of a container are usually given in form of a container definition file. This file is a text file containing different sections that describe the container build process. There is no standard used and as such these files can look slightly different. Mostly they contain the same sections and some minor differences but are incompatible between daemons, e.g. *Docker* vs *Singularity*. One reason why tools such as HPCCM are usefull.

2.1 HPC Container Maker

HPCCM [8] is developed by *NVIDIA* as an open source software package. It's purpose is to facilitate the process of creating container definition files. Achieved is this goal via a high level python coding interface. *HPCCM* provides predefined building blocks for common container building tasks. Additionally it provides modules for user specific commands [9]. The high level python container definition script can be translated into *Docker* or *Singularity* definition files for high flexibility. Container best practices -such as a cleanup in every run statement- are applied automatically. The usage of python is a benefit when creating *HPCCM* recipes as it provides a more programmable interface as pure *Docker* or *Singularity* definition files.

A very simple container definition file which shows the basic usage is shown below in listing 1.

Supposed the content from listing 1 is saved into a file called *mpi.py*, to compile it into a *Docker* definition file the following command can be used:
`hpccm --recipe mpi.py --format docker > Dockerfile.mpi`. Note that *HPCCM*

[7] `https://opencontainers.org/`

[8] https://github.com/NVIDIA/hpc-container-maker

[9] Via the flexible shell building block in which all commands are put into a usual container run statement it is possible to build arbitrary container definition files even if no *HPCCM* building block is available for this task.

```python
#!/usr/bin/python3
Stage0 += baseimage(image='docker.io/library/ubuntu:22.04')

# add mpi
Stage0 += openmpi(version='4.0.3')
```

Listing 1: *HPCCM* definition file which will install *Open MPI*. The resulting *Docker* container definition file can be seen in the appendix, listing 5.

outputs a string -usfull for using it inside scripts with variables- which is redirected into the file *Dockerfile.mpi*. The glazing content of this file can be viewed in the appendix (listing 5).

3 Methodology

In order to enable a user to use his own version of *GROMACS* [Bek+], which does not come preinstalled on the HPC system, a containerized approach can be chosen. For this Azat Khuziyakhmetov made a script which builds the container. The approach is a multi image build which means that to produce a final image not all software installation steps are made in one build step. Rather four different images are build on top of each other. The first image contains a installation of *Spack*. The second image uses the first image as base image and a specific version of *GNU Compiler Collection (GCC)*, *MPI* and *Fastest Fourier Transform in the West (FFTW)* utilizing *Spack* is installed. Using the second image as base image the third image installs a specific version of *GROMACS* using *Spack*. Thus the final image contains a working version of *GROMACS* with *MPI* support. However the final image is relatively large in size as it contains the compiler libraries. These are not needed to run simulations with *GROMACS*.

As solution the *GCC* compiler and all other build dependencies should be removed. Removal is however not trivial in the context of container images and their layers. Each *run* statement in a *Podman* or *Docker* container definition file will create a new layer in the image. The layer contains all changes made in respect to the previous layer. Hence if a file is removed the change -the removal of the file- is reflected in the corresponding layer. All subsequent layers will not contain the file. But the file is present in earlier layers. Removing a file in a layer will only affect the following layers it will not change any previous layers (see figure 1). As such the removed file will be inside the final image -the final image contains all layers- and contributing to its size. Shrinking the size of a container image is therefore not possible by simply removing files or uninstall software.

Different solutions exist to overcome the problem. Since a new layer is only made for each *run* command in the definition file, one solution is to pull all software installation and removal commands in one *run* statement. This would resolve the image size problem as there will be only one layer in the final image not containing any removed files. Unfortunately making changes or simply read the container definition file will be difficult. If the build process fails at a later stage in the run command there will be no cached layers

and as such the whole build process has to start from the beginning which can take a lot of time. As the underlying approach of this work is to build multiple images on top of each other it is not possible to combine all installation and removal steps into one run command.

Another approach is to start/run a container from the final image. Though all removed software is contained inside the image, once a container is instantiated it will have the state of the final layer of the image. As such all unnecessary software is not present. The instantiated container can be exported into a *tar* archive which can be imported as image again. Thus in the end the imported image will contain only one layer without the removed software. However exporting and importing a container takes time and space and is outside of the usual build process, utilizing a feature that war originally build for a different purpose.

A third option is to use a build and a deploy image. All the required software is build inside the build image and finally only the software binaries that are really needed will be copied to the deploy image. In the case of our multi stage build this would mean that the *GCC* compiler suite and all its dependencies as well as all other no longer needed software is removed from the *Spack* environment. Subsequently a new image is build which contains only a working *Spack* installation. In this image the cleaned *Spack* environment is copied, resulting in an image only containing the needed parts in its layers.

Since the problem arise from the layer stacking approach a last option is to crush all layers into one, thus representing only the final state. This can be achieved during the container build process by an argument that tells the builder to *squash* all layers. Hence very few changes need to be made into the build script and it is not necessary to find all relevant files that need to be copied to a deploy image as in the last approach. However *Docker* considers this feature as experimental since 2017 and in my testing it did not always work -the layers weren't squashed at all-. In contrast *Podman* does the job nicely and all layers are squashed into one final layer.

3.1 build script

Only the last two solutions are considered reasonable. To have the option to switch easily between them a easily to control build script should be available. Here one should be able to specify which of the two size reduction methods should be used and the container build process should be adapted accordingly.

In order to install a different version of the software library's it was initially necessary to change the container definition files accordingly. The build script should take care of this and the corresponding parameters should be changeable very easily at the beginning. Building a container with a specific version of *GCC*, *MPI*, *FFTW* and *GROMACS* should be simple then. Additionally the target architecture should be defined which will build a hardware optimised software stack.

In the end a reasonable sized container image should be automatically build and can be transferred to the HPC system for further usage.

3.2 Expected outcome

Without deep diving into containers and their build instructions it should be possible to generate a working *GROMACS* container image. It should be only very few knowledge about *Spack* install commands needed to define which software versions are be installed. Beside the usability effects the resulting container image should be as small as possible for convenient transfer to the HPC system and storage reasons. The performance of *GROMACS* should be reasonable compared to a bare metal installation.

4 Implementation

This section will describe some core implementation aspects in more detail. There are two main parts that are involved in the container build process. First the build script and secondly the *hppcm* recipe files. All of the driving logic is implemented in the build script. Here variables are defined which will control the build process based on their values (see appendix listing 2 for all variables) The script will copy *HPCCM* templates -residing in a folder called templates- into its directory. It will modify them via the *sed* [10] tool to add image names, software versions, image shrinkage method and the like. After this step the *HPCCM* recipes are ready and are compiled into container definition files. These definitions files will be then used to build the containers, where depending on the users setting *Docker* or *Podman* is used.

4.1 Deploy methods

In order to shrink the image size the compiler and other not needed software packages are removed.

Since the packages remain in the image layers below the current one in which the removal of the packages happens, the overall size of the image is not reduced as discussed in the last chapter. To overcome the problem there are two mentioned approaches implemented.

The simplest option is to build the last image with a flag that tells the build command to squash all layers in the image. In *Podman* the flag is named `--squash-all` in *Docker* `--squash`. Hence the final *hppcm* build recipe will only contain commands to remove the software as shown in listing 3, and compiled into a container definition file. The subsequent *Podman* image build instruction would be
"`podman build --squash-all -t img_name -f Dockerfile.deploy .`".

The other approach is to clean up the *Spack* environment, build a new and clean deploy image and copy over the cleaned environment. Additionally there is the need to set up Spack into the new image by adding spacks completion bash and adding *Spack* and its binary locations to the PATH variable. The corresponding *HPCCM* build script is shown in listing 4. Since the deploy image does not have layers with unwanted content in it the build is performed usually, without the `--squash-all` flag, and in case of *Podman* -or

[10] https://www.gnu.org/software/sed/

Docker- is
" `podman build -t DEPLOY_IMAGE_NAME -f Dockerfile.deploy .` ".

5 Evaluation/Results

The work described above significantly reduces the workload for a user to build a container image for *GROMACS*. With the developed build script it is not necessary to go into the different recipe files used by *HPCCM* in order to change the version or target architecture. All relevant variables appear in one place and an easy overview is possible. As such it is easy to see which exact software versions are installed.

The final image is much smaller compared to the initial build process since the compiler and other unused software is removed and are not present in any layers of the container. Sizes depend on the software versions installed as different *GCC* or *GROMACS* versions will differ in their space requirements. For an installation of the versions listed in table 1 the final image size was reduced from 4280 to 2070 MB which is a reduction of 51 percent.

image	size (MB)	software added (+) removed (−)
root	80	+ Ubuntu@22.04
base	596	+ Spack@0.19, updates(Ubuntu)
mpi	3910	+ GCC@12.2.0, OpenMPI@4.1.3, FFTW@3.3.10
gromacs	4280	+ GROMACS@2022.3
deploy	2070	− GCC@12.2.0, $libraries_{compile}$

Table 1: Sizes for the different container images generated during the build steps. Note that the software column includes the installation or removal of all needed dependencies -which are not listed-. The root image is not build, instead it is pulled from the *Docker* library at "*docker.io/library/ubuntu:22.04*".

It was unexpected that the `--squash` flag in *Docker* version 20.10.23 did not work. Building the image using it did not reduced the size of the final deploy image. To mention is that the flag is listed as experimental since 2017. However it worked really well in *Podman* and therefore *Podman* was adapted as the default builder.

Another important aspect is performance. Performance tests where made with the *GROMACS* benchmark from the Max Plank Institute [11], specifically *benchMEM* where used. Although testing was only made empirically the author can report that a performance impact due to containerisation was marginally. With different simulations and or systems the result might be different.

[11] `https://www.mpinat.mpg.de/grubmueller/bench`

6 Challenges / Discussion

Building container images can be a very compute heavy task. Since *Spack* is used which will compile all installed software from source -which is a good thing as then the appropriate compile flags and compiler optimisations can be performed- the container build is a very CPU hungry endeavour. On a Intel xeon $E3-1245V2$ the build for the containers specified in table 1 took roughly about 4 hours. Since compilation of software is not always a smooth process it can be very annoying to build the container images. It can require some trial and error before a working combination of software versions can be found. The multi image approach reduces the burden as not all images need to be rebuild on a failure. However it would be great if containers could be build on the potent HPC systems. This can be a non trivial task since the container build process might requires to run as *root* which is not possible on such systems. Rootless container daemons like *singularity* or *Podman* are therefore a step into the right direction.

Containers have their specialities when it comes to accessing the hosts file system. In *singularity* the user file system is mounted automatically and its files are accessible easily, in *Docker* and *Podman* the user has to explicitly mount the file system. As a new user one has to become used to such facts and how to use the software installed inside the container. This can be especially problematic if software within the container should run on a HPC system where is has to use schedulers like *slurm*. The experience showed that it is not easy as often permission or path problems occured.

Another observation is that *HPCCM* does not provide building blocks for *Spack*. As a work around the *shell* block can be used to issue the correct *Spack* commands. However it might be simpler to have a special *Spack* building block. This block could automatically apply best practices for *Spack*.

Spack itself offers a containerize option [12]. A *Spack* environment -defined in the *Spack* way in a *spack.yaml* file- can be automatically translated into a *Docker* definition file by issuing the command `spack containerize > Dockerfile`. It will build a container, based on a precompiled Ubuntu based *Spack* image, and installs the software defined in the *spack.yaml* file. Finally it will copy the environment into a new deploy image to reduce the final image size. While this is a nice feature of *Spack*, the multi image approach described above has advantages. It does not need a working *Spack* installation. It is more flexible, in particular it is possible to change the root image or install software beside the *Spack* environment. It generates reusable images, for example the MPI image can be used to install a different simulation software then *GROMACS* without the need of changing the underlying approach.

It is possible to run a *GROMACS* simulation using the compiled container images on the compute nodes of the HPC system. However the process might be not very smooth. If the container was started with *slurm*, some part of *Open MPI* tried to write into `/local` inside the container. This stops the simulation with an error since the container directories are not writable. As solution one can bind a writable directory into the same path into the container as described in the repository of this work. Another failure might happen if *Open MPI* tries to run over the network on more then one node, which is the case why

[12] `https://spack.readthedocs.io/en/latest/containers.html`

it is used in the first place. Here errors and aborted simulations occurred for reasons that where not really understood.

7 Conclusion

Containers provide a powerful environment for deploying software. They can be used by users of HPC systems to use their own installed and configured software without the need of having to ask the system administrator. However building and running containers especially on HPC systems can be a non trivial task as schedule managers and permissions might throw a shadow over the users experience. Although they are sometimes the only way of giving the user access to arbitrary software packages. And with a little experimenting it should be possible to resolve all problems and have a well running containerized simulation.

For me this course was the first time I worked with containers. I am very excited about the ability to provide a separated and save environment to run software yet do not overload the hardware resources. Having a container build script allows for easy installation of the selected software on different machines. This streamlines the way I will use software in the future. Additionally I learned how to use containers on a *HPC* system and submit a batch job to *slurm* that runs the installed software in the container on the compute nodes.

The source code and further usage instructions on how to run the container with *slurm* can be found in the corresponding repository [13].

[13] `https://gitlab.com/Winnus/gromacs-container-for-hpc`

References

[Bek+] H. Bekker et al. "“Gromacs: A parallel computer for molecular dynamics simulations”; pp. 252–256 in Physics computing 92. Edited by R.A. de Groot and J. Nadrchal. World Scientific, Singapore, 1993." In: ().

[Heo+18] Matt et. al Heon et al. *Podman - : A tool for managing OCI containers and pods*. Version v1.0 and beyond. Currently at v3.0.1. Jan. 2018. DOI: `10.5281/zenodo.4735634`. URL: `https://doi.org/10.5281/zenodo.4735634`.

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (May 2017), pp. 1–20. DOI: `10.1371/journal.pone.0177459`. URL: `https://doi.org/10.1371/journal.pone.0177459`.

A Appendix

B Figures

```
FROM docker.io/library/ubuntu:22.04

RUN touch myfile
RUN rm myfile
```

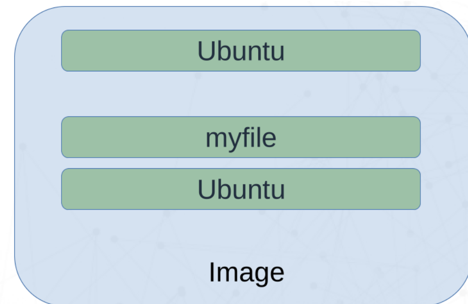

Figure 1: Schematic drawing of specific layers in a container image. Whenever a new run statement is made a new layer is added to the image. *myfile* is added into the second layer of the image due to the first run command. The second run command removes the file and as such it is not present anymore in the third layer of the image. However it is still present in layer two of the image and as such contributes to the overall size of the image.

C Code samples

```
## -------------------------------
##      variables
## -------------------------------
## define builder
BUILDER='podman' ## 'podman' 'docker'

## remove build recipes once build is finished?
REMOVE_RECIPES='false' ## 'true' 'false'

## define how to deploy the final image
## squash: builder removes all layers in final build
## copy: spack env will be copied to new, clean image
FINAL_IMAGE_DEPLOY_METHOD='squash' ## 'squash' 'copy'

## define the build architecture
ARCH='cascadelake' ## 'cascadelake' 'ivybridge' 'hashwell' ...

## image names
ROOT_IMAGE_NAME='docker.io/library/ubuntu:22.04'
BASE_IMAGE_NAME="gwdg/hpc-base-ubuntu-$ARCH:latest"
MPI_IMAGE_NAME="gwdg/hpc-mpi-ubuntu-$ARCH:latest"
GROMACS_IMAGE_NAME="gwdg/hpc-gromacs-ubuntu-$ARCH:latest"
DEPLOY_IMAGE_NAME="gwdg/hpc-deploy-ubuntu-$ARCH:latest"

## spack version
SPACK_VERSION='v0.19'

## declare compiler, mpi and fftw to be installed
## these have to be spack commands
## to get available versions use a spack installation or
## use their website:
## "https://spack.readthedocs.io/en/latest/package_list.html"
GCC_VERSION='gcc@12.2.0 target=x86_64'
MPI_VERSION='openmpi@4.1.3' ## do not provide GCC version (added automatically)
FFTW_VERSION='fftw' ## do not provide the architecture (target=xyz) here
GROMACS_VERSION='gromacs@2022.3'
```

Listing 2: Variable section of the build script containing all variables that can be modified.

```python
#!/usr/bin/python3
Stage0 += baseimage(image={{GROMACS_IMAGE_NAME}}, _distro='ubuntu22')
gcc_version = USERARG.get('gcc_version', {{GCC_VERSION}})

# clean up:
# clean gcc and clean env
Stage0 += shell(commands=
        [
            'spack uninstall -y {}'.format(gcc_version),
            'spack gc -y',
            '. /load-spack-env.sh',
            'spack gc -y'
        ])
```

Listing 3: HPCCM recipe to remove the compiler and software garbage with spack. A shell command is specified which: uses spack to uninstall the compiler from the system wide packages, doing a garbage collection on the system wide installed spack files, loads the spack environment in which *gromacs* is installed, doing a garbage collection inside that environment.

```
#!/usr/bin/python3
# -------------------------------------------
# clean up
# -------------------------------------------
Stage0 += baseimage(image={{GROMACS_IMAGE_NAME}}, _distro='ubuntu22',
_as='base')

gcc_version = USERARG.get('gcc_version', {{GCC_VERSION}})

# clean gcc and clean env
Stage0 += shell(commands=
        [
            'spack uninstall -y {}'.format(gcc_version),
            'spack gc -y',
            '. /load-spack-env.sh',
            'spack gc -y'
        ])

# -------------------------------------------
# copy spack env to new base
# -------------------------------------------
Stage1 += baseimage(image={{ROOT_IMAGE_NAME}}, _distro="ubuntu22")

# Spack dependencies and Python
ospackages = ['build-essential', 'make', 'patch', 'bash', 'tar', 'gzip',
'unzip', 'bzip2', 'xz-utils', 'zstd', 'file', 'gnupg2', 'git',
'python3-dev', 'curl', 'ca-certificates', 'autoconf', 'vim',
'pkg-config', 'gfortran', 'python2', 'python3'
]
Stage1 += apt_get(ospackages=ospackages)

# Copy Spack
Stage1 += copy(_from='base', src='/opt', dest='/opt')
Stage1 += copy(_from='base', src='/root/.spack', dest='/root/.spack')
Stage1 += shell(commands=[
    'ln -s /opt/spack/share/spack/setup-env.sh /etc/profile.d/spack.sh',
    'ln -s /opt/spack/share/spack/spack-completion.bash /etc/profile.d'
])
Stage1 += environment(variables={'PATH':'/opt/spack/bin:/opt/view:$PATH',
'SPACK_ROOT': '/opt/spack'})
Stage1 += shell(commands=[
    'echo ". /opt/spack/share/spack/setup-env.sh" >> /load-spack-env.sh',
    'echo "spack env activate /opt/spack-env" >> /load-spack-env.sh'
])
```

Listing 4: HPCCM recipe to remove the compiler and software garbage with spack and copy the cleaned *spack* environment into freshly build deploy image. *Spack* specific setups are done as well.

```
FROM docker.io/library/ubuntu:22.04

# OpenMPI version 4.0.3
RUN apt-get update -y && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y \
        --no-install-recommends \
        bzip2 \
        file \
        hwloc \
        libnuma-dev \
        make \
        openssh-client \
        perl \
        tar \
        wget && \
    rm -rf /var/lib/apt/lists/*
RUN mkdir -p /var/tmp &&
    wget -q -nc --no-check-certificate -P /var/tmp \
    https://www.open-mpi.org/software/ompi/v4.0/downloads/\
    openmpi-4.0.3.tar.bz2 && \
    mkdir -p /var/tmp && \
    tar -x -f /var/tmp/openmpi-4.0.3.tar.bz2 -C /var/tmp -j && \
    cd /var/tmp/openmpi-4.0.3 && \
    ./configure --prefix=/usr/local/openmpi --disable-getpwuid \
    --enable-orterun-prefix-by-default --with-cuda --with-verbs && \
    make -j$(nproc) && \
    make -j$(nproc) install && \
    rm -rf /var/tmp/openmpi-4.0.3 /var/tmp/openmpi-4.0.3.tar.bz2
ENV LD_LIBRARY_PATH=/usr/local/openmpi/lib:$LD_LIBRARY_PATH \
    PATH=/usr/local/openmpi/bin:$PATH
```

Listing 5: Docker definition file to install *openmpi* in a container. The definition file was automatically generated by *HPCCM* using the recipe listed in 1