Apptainer on Claix – An Introduction

Agenda

- Why Containers?
- Apptainer: Containers for HPC
- Build Containers
- Container Best Practices
- Apptainer and MPI

Why Containers?

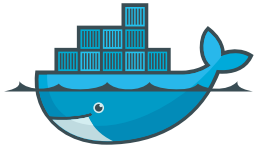
Why Containers?

- Migration of software often hasslesome
  - Target system may not support all dependencies
  - … or only in incompatible versions
- Installation of large software components on host may be unwanted
- Partial or complete isolation from the host system may be wanted
- Reproducibility may be a concern!
- Virtual machines are powerful but …
  - … are resource-heavy
  - … virtualize too much
  - … are not available to all users

Image sources:
https://www.docker.com/company/newsroom/media-resources/
https://github.com/apptainer/apptainer-logos/tree/main
https://github.com/containers/podman.io/blob/main/static/logos

What are Containers?

- Lightweight virtualization technique
- Containers virtualize parts of the host operating system via „namespaces“, allowing for
  - Different user permissions
  - Different network settings
  - Different filesystem views
  - etc.
- They introduce no significant runtime overhead!



Apptainer: Containers for HPC

Apptainer Versus Other Runtimes

- HPC systems work different from local workstations
- Full isolation is counter-productive to parallel jobs
- Daemon-based runtimes do not work well with workload managers (Slurm)
- Apptainer has been designed with HPC in mind
  - isolates as little as necessary by default

	Network Defaults	Filesystem Defaults	GPU Support	Network Fabric Support	Execution
Docker	Bridged containers	Isolated	Requires ext. tools	With configuration	Via docker daemon
Podman	Bridged containers	Isolated	Requires ext. tools	With configuration	Self-contained process
Apptainer	Host network	Home directory available	Native (CUDA, ROCm)	Out of the box	Self-contained process

Apptainer: Key Features

- Rootless workflows
  - Build, download and run images as regular user
  - No additional permissions required
- Support for Docker images
  - Pull or run images from public registries (Docker Hub) in a single command
  - Use Docker images as base to bootstrap own Apptainer images
- Images can be run like regular programs
  - … which simplifies batch scripts a lot
- Image format supports signing and encrypting contents
  - Ensure integrity and confidentiality of contained data and programs
  - Interesting if you need to process highly confidential data records

Build Containers

Apptainer Definition Files

- Images contain everything required for the containerized applications
  - Minimal operating system
  - Program code
  - Maybe data or input files
- **Definition files** describe how an image should be built!
- Setting up the software environment becomes a matter of a single command
- **Good news**: Images can be built either on your machine or on the cluster

Definition Files

- Bootstrap option defines where to pull the image base from
  - DockerHub images work out-of-the-box
  - Other images can also serve as base
- File is grouped by `%sections`
  - Covers everything from post-setup commands to environment and metadata
- Images can be explored with `shell` command
- Runscript definition allows for user-friendly startup
  - … and much simpler batch scripts

```
Bootstrap: docker
From: rockylinux:8

%post
    echo "Installing required packages..."
    printf '%s\n' 'assumeyes=true' 'max_parallel_downloads=15' >> /etc/dnf/dnf.conf
    dnf update
    dnf group install "Development Tools"
    dnf install wget
    dnf clean all

    echo "Here, we would install the actual application"

%runscript
   echo "These were your arguments:"
   exec echo "$@"
```

```
$ apptainer build example.sif example.def
...
$ ./example.sif This is simple
These were your arguments:
This is simple
```

```
$ apptainer shell example.sif
# Now we work within the container
Apptainer> gcc --version
gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-22)
```

Documentation:
https://apptainer.org/docs/user/main/definition_files.html

Best Practices

- Containers should be **stateless**
  - The image only changes with version updates etc.
  - Extra configuration and input files are passed in at runtime
    - Bindpaths allow for clean configuration management
  - If you want to change something in an existing image, you are probably doing something wrong!
    - Rebuild it with an updated definition file!
- Keep definition files **small** and clean package manager caches
  - Small image footprints improve performance
  - If your applications do not need it, it should not be part of your image
- Bootstrapping your own base images simplifies building pipelines
  - Save time by re-using the same base OS
- Save your images as SIFs on the `$WORK` filesystem
  - They do not need to be backed up (unlike definition files)
  - Sandbox directories are harder to manage and put more stress on the filesystem

Apptainer and MPI

Apptainer and MPI – The Basics

- Containerized MPI can be tricky!

- Ideally the containerized MPI is compatible with the host software stack
  - Neat integration with Slurm‘s MPI features
  - Less user setup required

- **But**: Containers are supposed to be portable
  - Creators cannot target all possible HPC centers out there
  - Application might rely on specific MPI stack
  - Packaging host-dependent stacks in containers seems pointless…

- **Solution**: Choose execution model based on the use case

Apptainer and MPI – The Models

- Execution models differ in
  - Setup complexity
  - Flexibility
  - Host compatibility

- **Hybrid model**
  - Process management through host MPI
  - Application is linked against MPI inside container
  - Requires ABI-compatible MPI versions on host and inside container

- **„Containerized“ Model**
  - Containerized MPI and PMI implementations
  - The entire stack is part of the container
  - Container needs support for host hardware

- **Bind model**
  - Container comes without MPI
  - Host MPI is bind-mounted into the container at runtime
  - Binaries need to be linked against and run with the bind-mounted runtime

Apptainer and MPI – The Models

Hybrid Model

- \+ Works with foreign containers
- \+ Integrates with Slurm
- − Requires matching MPIs on both sides

Containerized Model

- \+ Most portable solution
- \+ Full control over used MPI implementation
- \+ Images also usable for hybrid model
- − Requires proper base images or understanding of MPI

Bind Model

- \+ Provides MPI support for *exotic* use cases
- − Application is fully host-dependent
- − Configuration is convoluted and error-prone

Example: Containerized Model

We want to run a „Hello world“ example.

1. Download and build OpenMPI 4.1.4
   - Build is very rudimentary
2. Include it in our container environment
   - Make the application runnable with no further runtime configuration
3. Build our example program using the containerized MPI
   - This is no different from building it on the host

```
Bootstrap: docker
From: rockylinux:8

%files
    helloworld.c /usr/local/share

%arguments
    OMPI_VER=4.1.4

%environment
    # Point to OMPI binaries, libraries, man pages
    export OMPI_DIR=/opt/ompi
    export PATH="$OMPI_DIR/bin:$PATH"
    export LD_LIBRARY_PATH="$OMPI_DIR/lib:$LD_LIBRARY_PATH"
    export MANPATH="$OMPI_DIR/share/man:$MANPATH"

%post
    printf '%s\n' 'assumeyes=true' 'max_parallel_downloads=15' >> /etc/dnf/dnf.conf
    dnf update
    dnf group install "Development Tools"
    dnf install wget
    dnf clean all

    export OMPI_DIR=/opt/ompi
    export OMPI_VERSION={{ OMPI_VER }}
    export OMPI_URL="https://download.open-mpi.org/release/open-mpi/v${OMPI_VERSION:0:3}/openmpi-
$OMPI_VERSION.tar.bz2"
    mkdir -p /tmp/ompi
    mkdir -p /opt
    # Download
    cd /tmp/ompi && wget -O openmpi-$OMPI_VERSION.tar.bz2 $OMPI_URL && tar -xjf openmpi-$OMPI_VERSION.tar.bz2
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure --prefix=$OMPI_DIR && make -j8 install

    # Set env variables so we can compile our application
    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:$LD_LIBRARY_PATH

    echo "Compiling the MPI application..."
    cd /usr/local/share && mpicc -o /usr/local/bin/mpihello helloworld.c

%runscript
   echo "Running MPI Hello world example!"
   exec mpiexec "$@" /usr/local/bin/mpihello
```

Example based on: https://apptainer.org/docs/user/main/mpi.html#open-mpi-hybrid-container

Example – Containerized Model

- Our runscript is simple but very flexible
  - We set the framework, the user provides the input
  - In real scenarios, let mpiexec do its magic
- The entire configuration is containerized
  - Container is invoked like any other application
  - Using applications directly = less confusion for inexperienced users
- The OpenMPI version is also available on the host
  - Hybrid execution is one module call away
  - This example also works with Slurm‘s `srun`

To avoid problems, run
`unset -m 'PMIX_*'`
first!

```
# We need no further arguments
$ ./hello.sif
Hello, I am rank 1/8
Hello, I am rank 2/8
Hello, I am rank 3/8
Hello, I am rank 4/8
Hello, I am rank 5/8
Hello, I am rank 6/8
Hello, I am rank 7/8
Hello, I am rank 0/8
# Now, let's pass a process number to mpiexec
$ ./hello.sif -n 4
Hello, I am rank 3/4
Hello, I am rank 2/4
Hello, I am rank 0/4
Hello, I am rank 1/4
```

Containerized Execution example

```
# Hybrid execution works, too!
$ ml purge; ml foss/2022a
$ mpiexec -n 4 apptainer exec hello.sif /usr/local/bin/mpihello
Hello, I am rank 2/4
Hello, I am rank 3/4
Hello, I am rank 1/4
Hello, I am rank 0/4
```

Hybrid execution example

Outlook: Claix Container Tree

- Proper configuration of MPI stacks requires knowledge
  - Researchers have better things to do than fiddle with technicalities
  - Debugging MPI issues can be time-consuming and frustrating

- **Solution:** Tree of Claix-specific images
  - Part of an NHR effort to aid in HPC container development
  - Provide toolchains tried and tested on our host systems
  - Pre-configured for optimized performance
  - Give you a sane image collection to deploy the software *you* care about

- Coming this year!

Thank you for your attention!