Hewlett Packard
Enterprise

CONTAINERS IN HPC

Alfio Lazzaro & Harvey Richardson
HPE HPC/AI EMEA Research Lab

ARCHER2 Webinar, March 2021

OUTLINE

- Brief introduction to Containers
  - Why they are useful
  - Docker: the *de-facto* industrial standard technology

- Containers in HPC
  - HPC requirements
  - Specific solutions: Singularity, Charliecloud, Sarus, Shifter

- HPC in Containers
  - Performance challenges
  - Cross-compilation
  - Optimized libraries
  - Hardware support, eg accelerators and network
  - MODAK: Management of Optimized Deployment of Applications with Containers

CAVEATS (1)

- This is not a tutorial on how to use containers
  - Based on (our) user experience, ie no admin experience
    - Ignore installation, management, and security aspects

- Only focus on HPC systems
  - Batch system compatibility (eg SLURM and PBS)
  - Support for optimized libraries and hardware, eg MPI and GPU supports

- This presentation is relevant to containers in general
  - Although the examples will be based on Singularity
  - Some extra information is included to illustrate useful singularity commands and workflow

CAVEATS (2)

- Focus on how to achieve runtime performance
  - Ignore other aspects like formats, image size, building time…
  - However, we will not show performance results, rather we will concentrate on techniques to achieve performance
- Several reports reporting on performance of containerized applications, e.g.
  - A. Torrez, T. Randles and R. Priedhorsky, *HPC Container Runtimes have Minimal or No Performance Impact*, 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), Denver, CO, USA, 2019, pp. 37-42, doi: 10.1109/CANOPIE-HPC49598.2019.00010.

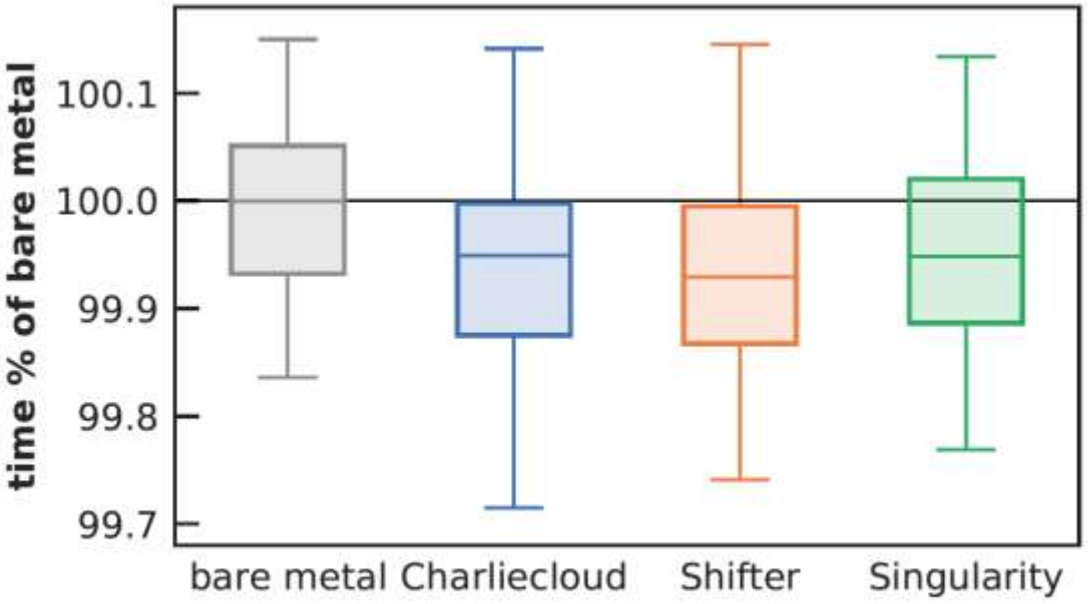


Fig. 1. SysBench prime number computation time relative to median bare metal performance of 129.36 seconds; lower is better. Boxes show the median and middle 50%, while whiskers show the maximum and minimum. The four environments showed essentially identical performance.

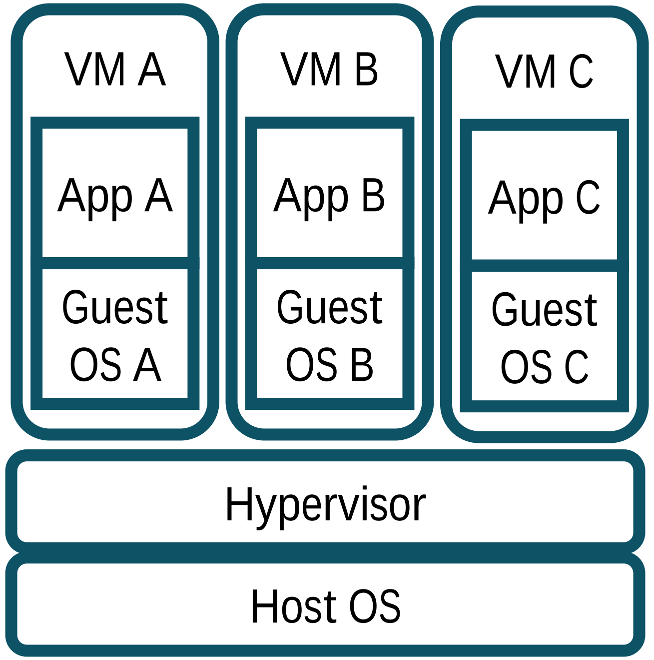
INTRODUCTION TO CONTAINERS

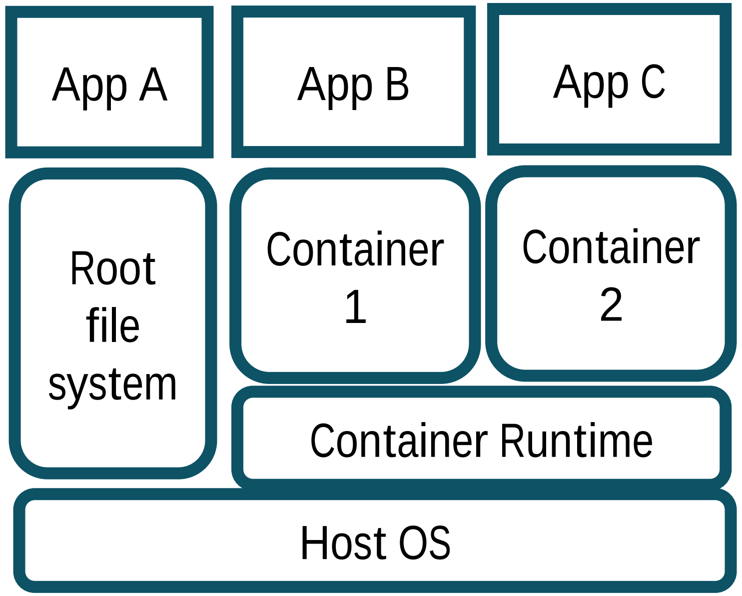
WHY CONTAINERS

- A technology to package and deploy software that runs with access to a limited set of host resources
  - Containers run in an isolated environment
- They solve the problem of making your software run reliably when moved from one computing environment to another (portable, reproducible workflow)
  - Using containers allow application deployment across systems without having to build and configure separately: Linux OS + your applications + all their dependencies, libraries and other binaries and configuration files needed to run, everything bundled in one package

CONTAINERS VS VIRTUAL MACHINES

- In contrast to virtual machines (VM), which virtualize the hardware and need a complete operating system, containers interface directly with the host’s Linux kernel, so they are faster to deploy and run
  - Clearly, containers can run within VMs (eg on Cloud)
- Containers need to interact with the host OS and are delivered as an image
  - Images are of a particular format and are generally configured from structured files





DOCKER (HTTPS://WWW.DOCKER.COM/)

- Designed primarily for network micro-service virtualization
- Dockerfiles are the *de-facto* standard for defining images (recipe files)
- Requires a daemon
  - Some issues with security – daemon running as root
    - Within the container you have root privileges
    - This is OK if you are running within a VM that you own
- Docker Hub is a service provided by Docker for finding and sharing container images (https://hub.docker.com/)
  - Over 100k container images from software vendors, open-source projects, and the community
  - Some examples:
    - AI frameworks: TensorFlow, Pytorch
    - Databases: MySQL, PostgresSQL
- Use the available containers as base to build your own containers

CONTAINERS IN MORE DETAIL

- A technology to package and deploy software that runs with access to a limited set of host resources.
- Namespace kernel feature used to achieve isolation of resources
- Process/task tree local to container
- Private and imported filesystems
- Networking capability
- Containers can be packaged as an image.
- Container images are of a particular format and are generally configured from structured files (yaml for example). Often need to be root to create the image.
- We think of the running instance as **a container**

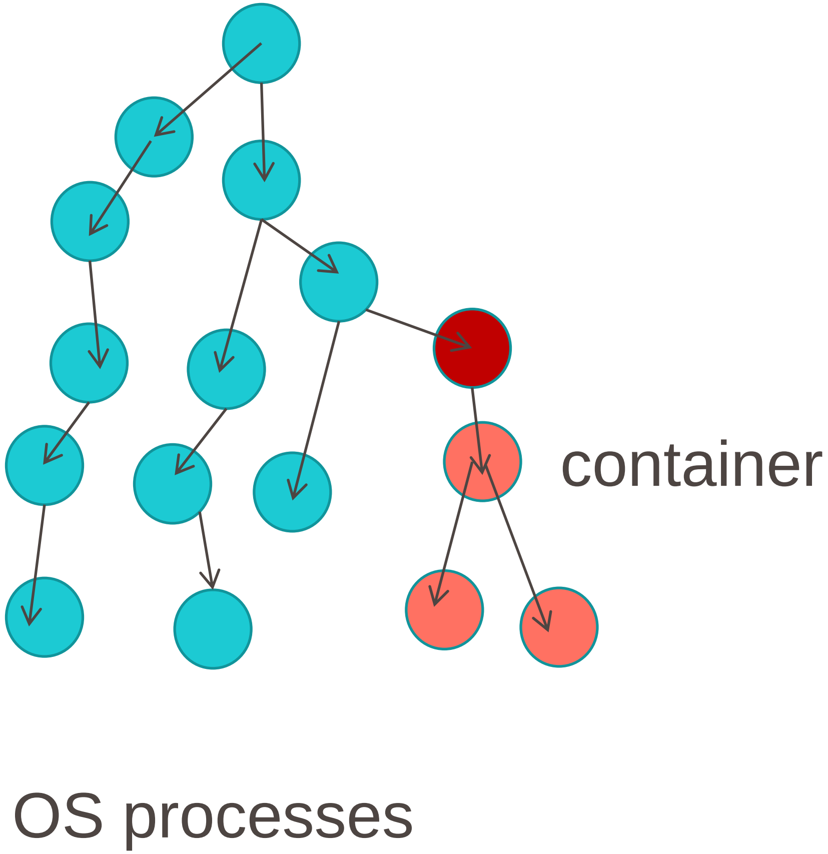
LINUX FEATURES THAT SUPPORT CONTAINERS

- Linux **Namespaces** are the core feature that isolates resources for containers

Proposed and **Implemented** namespaces (activated on clone, unshare, setns)

- **Mount (mounts)**
- **UTS (hostname, domain)**
- **PID (Process Ids)**
- **NET (Network)**
  - **RDMA**
- **IPC**
- **USER (user and groupid)**
- Cgroup
- Time (clocks)

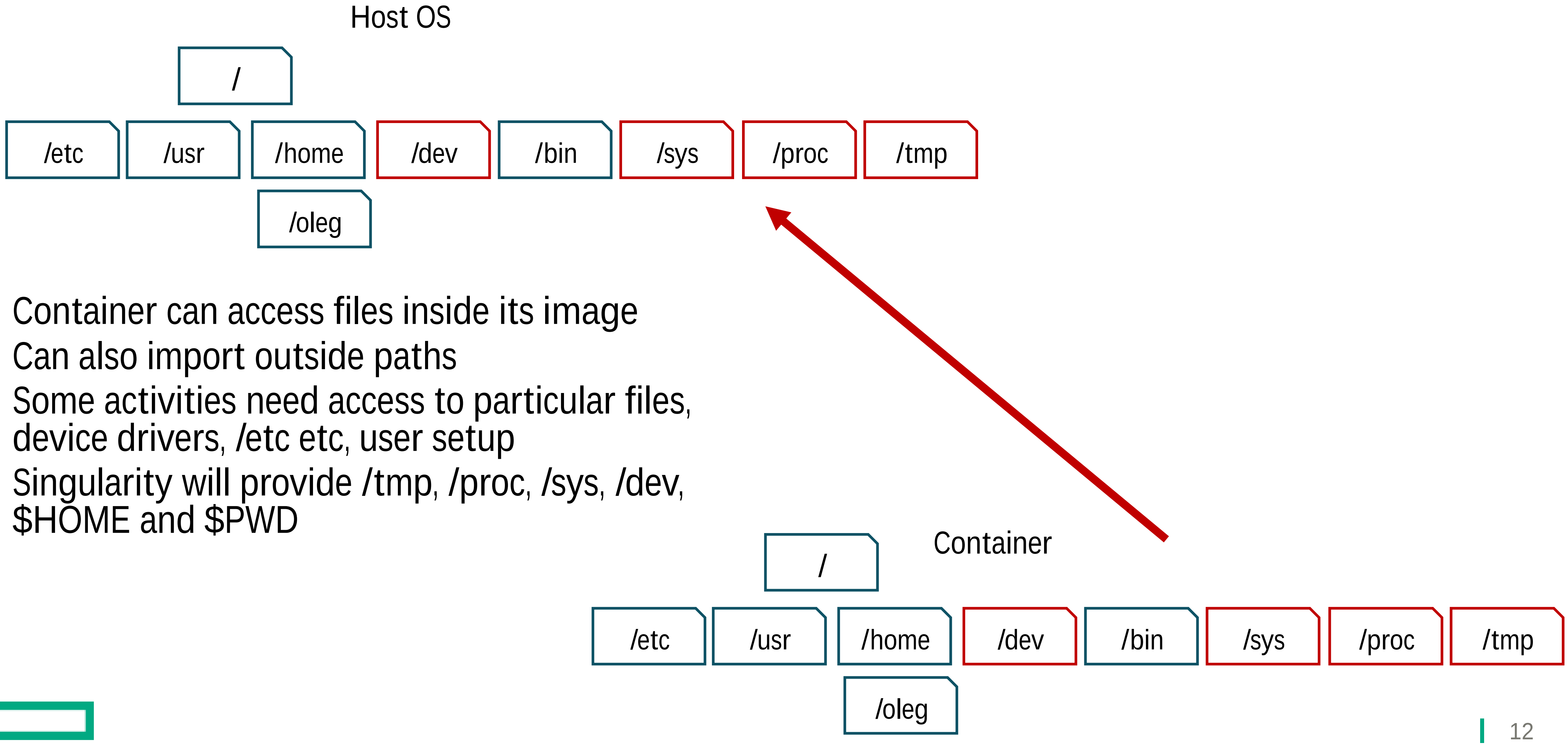
CONTAINERS AND THEIR ENVIRONMENT



The container processes can ‘see’ a different environment than in the host:

- Own system and developer software stack
- Own application software

CONTAINERS AND THEIR ENVIRONMENT



- Container can access files inside its image
- Can also import outside paths
- Some activities need access to particular files, device drivers, /etc etc, user setup
- Singularity will provide /tmp, /proc, /sys, /dev, $HOME and $PWD

THE OPEN CONTAINER INITIATIVE (OCI)

- Established in June 2015 by Docker and other leaders in the container industry, the OCI currently contains two specifications:
  - The Runtime Specification
    - The Runtime specifies 5 must-have API calls: Create, Start, Kill, Delete, Query state
    - The Runtime Specification defines an interface to plug-in, or hook, external programs to customize the container
  - The Image Specification
    - The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run
- Building containers is still a per-system function
  - OCI does NOT “do” building!

CONTAINERS IN HPC

HPC REQUERIMENTS

- No ROOT access and deamon
  - No privilege escalation
  - You do not want an end user to have root privilegies on a Supercomputer...
- Docker images compatibility (due to wide Docker adoption)
- Integration with workload managers, e.g. SLURM and PBS
- Support for diskless nodes, parallel filesystem friendly
- Support for hardware-dependent performance optimization
  - Network, Accelerators (GPU), CPU architectures...
    - Example 1: I compile my APP in the container on an AMD CPU with AVX2 vector instructions, then I run on an Intel CPU with AVX512 vector instructions
    - Example 2: I can compile my APP with MPI on an InfiniBand network, then I run on a HPE system with the Slingshot network
- Support for vendor optimized libraries and tools, e.g. scientific libraries and compilers

PORTABILITY & REPRODUCIBILITY VS PERFORMANCE

- Containers are meant for portability and reproducibility
  - Built in a system, run anywhere (from your laptop to Clouds and Supercomputers)
- But for performance you have to “customize” the containers
  - Breaks portability

(SOME) HPC CONTAINER SOLUTIONS

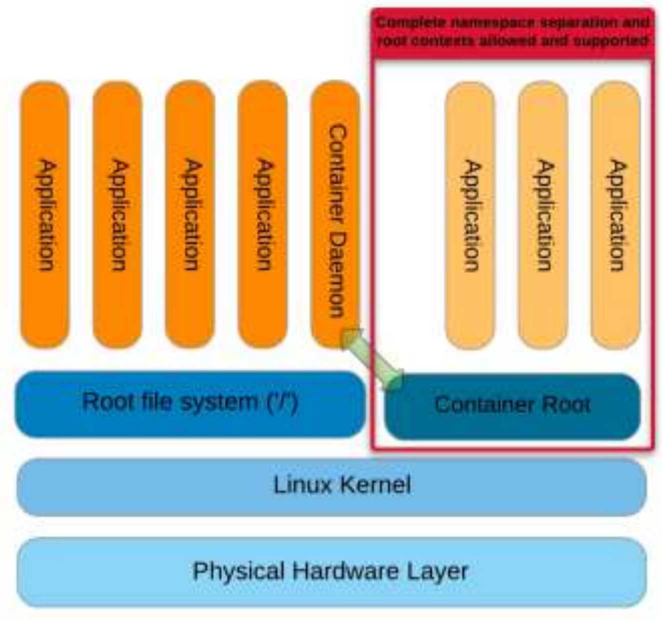
- Building and executing:
  - Singularity, https://github.com/sylabs/singularity
    - Building containers, recipe file syntax not compatible with Docker
    - Singularity Image Format (SIF) for the images
    - Can run Docker images

- Execute only (natively run Docker images):
  - CharlieCloud, https://github.com/hpc/charliecloud
  - Shifter, https://github.com/NERSC/shifter
  - Sarus, https://github.com/eth-cscs/sarus

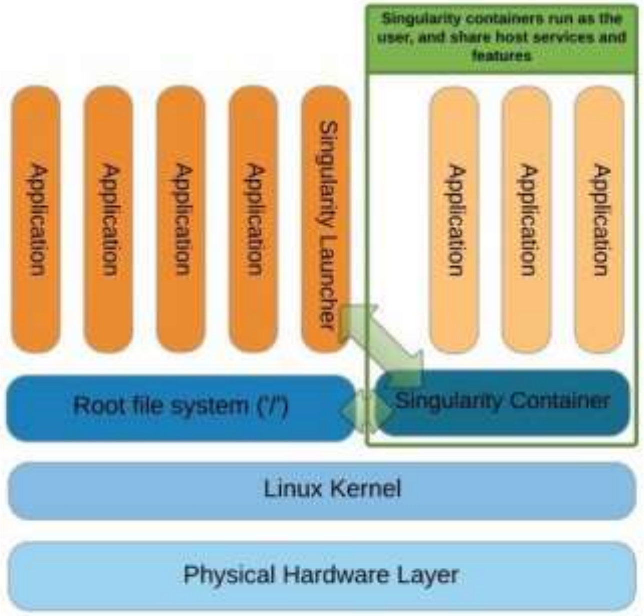
Sarus @ CSCS

- Common features:
  - Can run Docker images (Open Container Initiative compliance)
  - Rootless execution
  - Still require root privileges for building

SINGULARITY VS DOCKER: PRIVILEGES DESIGN



Docker



Singularity

- Singularity launches the container as the calling user in the appropriate process context
- There is no root daemon process and no escalation of privileges within the container
  - Limits user's privileges (inside user == outside user)

SOME SINGULARITY COMMANDS

Description	Command	Details
Version	`singularity version`	
Help	`singularity -h`	
Help on a specific command	`singularity help <command>`	
Manage OCI containers	`sudo singularity oci <command>`	Open-Containers-Initiative (https://www.opencontainers.org/): standardize containers management
Run an image	`singularity run <image>`	
Pull and run an image from Dockerhub	`singularity run docker://<image>`	E.g.: `singularity run docker://godlovedc/lolcow`
Exec a command within a container	`singularity exec <image> <command>`	
Open a shell within a container	`singularity shell <image>`	
Build a SIF image	`sudo singularity build <name>.sif <definition file>`	Singularity-Image-Format: compressed read-only format suitable for production
Build a sandbox image	`sudo singularity build --sandbox <name> <definition file>`	Writable (ch)root directory called a sandbox for interactive development

SINGULARITY DEFINITION FILE PARTS

- Header part where we set the parent images, eg

```
Bootstrap: docker
From: debian:stretch
```

- Sections, eg

Name	Short Description	Docker Corresponding Command
`%files`	Copies files from the host to the container, creates directories if needed	`COPY`
`%environment`	Allows you to define environment variables that will be set at runtime. **Overrides host variables**	`ENV`
`%post`	Executes commands during the building time	`RUN`
`%runscript`	Provides defaults for an executing container	`CMD`

DBCSR SINGULARITY EXAMPLE (1)

```
# Import parent image
Bootstrap: docker
From: debian:stretch

%files
    # Create destination directory and copy the performance test
    input.perf /workdir/dbcsr_bench/
```

DBCSR SINGULARITY EXAMPLE (2)

```
%post
    # Install general packages
    apt-get update && apt-get -y upgrade
    apt-get -y install --no-install-recommends build-essential wget file git ca-certificates \
      gfortran python libopenblas-dev && rm -rf /var/lib/apt/lists/*

    # Install MPICH
    export MPICH_VERSION=3.3.1
    wget -q http://www.mpich.org/static/downloads/${MPICH_VERSION}/mpich-${MPICH_VERSION}.tar.gz
    tar xf mpich-${MPICH_VERSION}.tar.gz && rm mpich-${MPICH_VERSION}.tar.gz
    cd mpich-${MPICH_VERSION}
    ./configure --prefix=/usr/local --disable-static --disable-rpath --disable-wrapper-rpath \
                      --mandir=/usr/share/man --enable-fast=all,O3
    make -j$(getconf _NPROCESSORS_ONLN) install && ldconfig && cd .. && rm -rf mpich-${MPICH_VERSION}

    # Install latest cmake
    export CMAKE_VERSION=3.15.3
    wget https://github.com/Kitware/CMake/releases/download/v${CMAKE_VERSION}/cmake-${CMAKE_VERSION}-Linux-x86_64.sh
    sh cmake-${CMAKE_VERSION}-Linux-x86_64.sh --prefix=/usr/local --skip-license && ldconfig
    rm -f cmake-${CMAKE_VERSION}-Linux-x86_64.sh

    # Compile DBCSR (https://github.com/cp2k/dbcsr) and copy the performance test
    cd /workdir/dbcsr_bench
    git clone --recursive https://github.com/cp2k/dbcsr.git
    cd dbcsr && mkdir build && cd build
    cmake -DUSE_MPI=ON -DUSE_OPENMP=ON -DCMAKE_BUILD_TYPE=Release ..
    make -j$(getconf _NPROCESSORS_ONLN) && cp -r tests/dbcsr_perf ../../ && cd ../../ && rm -rf dbcsr
```

DBCSR SINGULARITY EXAMPLE (3)

```
%environment
    # Default values if the variable are not previously declared on the host
    export NPROCS=${NPROCS:-1}
    export OMP_NUM_THREADS=${OMP_NUM_THREADS:-1}

%runscript
    # Default command, running inside the container
    # singularity run <image_name>
    # singularity exec <image_name> <command>
    mpirun -np ${NPROCS} /workdir/dbcsr_bench/dbcsr_perf \
                         /workdir/dbcsr_bench/input.perf
```

BUILDING THE IMAGE

- An image can be a single file or it can be useful to build into a set of files within a directory called a sandbox

- Typical way to build an image
  - `sudo singularity build dbcsr.sif dbcsr.def`
  - `sudo singularity build --sandbox dbcsr.imgdir dbcsr.def`

RUNNING THE CONTAINER

- SIF image can be directly executed
  - `./dbcsr.sif`
- More in general
  - `singularity run dbcsr.sif`
  - `singularity run dbcsr.imgdir`

- Setting MPI ranks and threads can be done on the host, e.g.
  - `NPROCS=2 OMP_NUM_THREADS=2 singularity run dbcsr.sif`
  - This is for running MPI inside the container, we will discuss how to run host MPI later on:

```
srun -n 2 singularity exec /workdir/dbcsr_bench/dbcsr_perf \
                            /workdir/dbcsr_bench/input.perf
```

INTERACTING WITH THE CONTAINER

- All commands valid for SIF and Sandbox images
- Execute a command, eg
  - ➢ `singularity exec dbcsr.sif whoami`
- Open a shell
  - ➢ `singularity shell dbcsr.sif`
- Notes
  - Same users between the host and the container
    - Singularity blocks privilege escalation, ie. no sudo inside the container
  - Same starting directory of the host
  - Some host directories automatically mounted (eg home directory)
    - Can use `-B` to bind more directories

CONATINER FOR PYTHON3/GTK+

- Challenge was to run a tool that needed python bindings for GTK+
- This needs both
  - Installation of rpms
  - Installation of python modules

CONATINER FOR PYTHON3/GTK+ (CONTAINER DEFINITION FILE)

```
# python3 container for python GTK+
BootStrap: docker
From: python:latest

%labels
Author email@xxx.yyy
Version v0.0.1
Description python container python3 GTK+ scripts

%post
# Install the necessary packages (from repo)
apt-get update && apt-get install -y --no-install-recommends \
 python3-numpy python3-gi python3-gi-cairo gir1.2-gtk-3.0 \
  libcanberra-gtk3-module \
  libgirepository1.0-dev gcc libcairo2-dev pkg-config python3-dev
apt-get clean

#Python packages
pip3 install numpy
pip3 install Pycairo
pip3 install PyGObject
```

CONATINER FOR PYTHON3/GTK+ (CONTAINER DEFINITION FILE)

```
mkdir -p /sv/

cat <<EOF >/sv/hello.py
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

window = Gtk.Window(title="GTK Example window")
window.show()
window.connect("destroy", Gtk.main_quit)
Gtk.main()
EOF
%environment
# Stop (most) dconf warnings
export DCONF_PROFILE=/tmp/disable_dconf
```

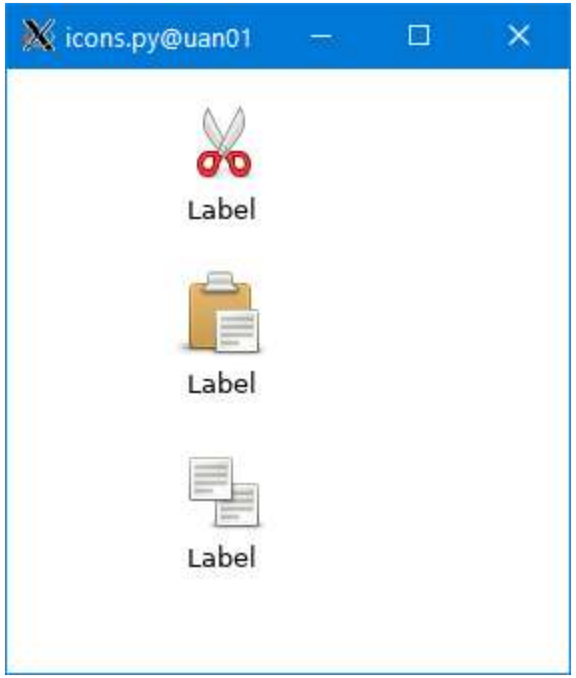
BUILD, TEST, USE

On build system

```
sudo singularity build python-gtk.sif python-gtk.def
singularity exec python-gtk.sif python3 /sv/hello.py
```

On deployment system

```
singularity exec python-gtk.sif python3 examples/icons.py
```



USING A BASE CONTAINER

- It can take a while to build a container and add packages to it, mistakes can be frustrating
- You can base a container on another.
- This example adds the IMB benchmark to a base container with MPICH installed

```
Bootstrap: localimage
From: mpich.sif

%post
export PATH=$PATH:/opt/mpich-3.3.2/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/mpich-3.3.2/lib

cd /tmp && wget https://github.com/intel/mpi-benchmarks/archive/IMB-v2019.6.tar.gz && tar zxvf IMB-v2019.6.tar.gz && cd mpi-benchmarks-IMB-v2019.6/src_c && make -j2 TARGET=MPI1 && install IMB-MPI1 /usr/local/bin/IMB-MPI1
```

USING THE ARCHER2 MPICH

- You will need to set appropriate library paths and bind host paths into a container to do this
- For example, in a batch script…

```
export SINGULARITYENV_LD_LIBRARY_PATH=/opt/cray/pe/mpich/8.0.16/ofi/gnu/9.1/lib-abi-mpich:/opt/cray/pe/lib64:/opt/cray/pe:/opt/cray/libfabric/1.11.0.0.233/lib64:/usr/lib64

 srun -n 128 singularity exec \
   --bind /opt,/etc,/var,/usr/lib64 mpich-imb.sif /usr/local/bin/IMB-MPI1 PingPong > IMB.out.Sing
```

Importing /usr/lib64 is a lot but otherwise the bind list will be quite long.

SINGULARITY INSTALLATION ON WINDOWS AND MACOS

- Singularity requires Linux as the host system!
  - Use a virtualized Linux, eg VirtualBox (https://www.virtualbox.org/)
  - Recommended Vagrant to install and manage a minimal Linux box (https://www.vagrantup.com/)
    - Better to include also Vagrant Manager (http://vagrantmanager.com/) to manage Vagrant VMs
  - Require a Shell, eg Git Bash on Windows (https://git-for-windows.github.io/)
  - Install Singularity on the Linux VM

HPC IN CONTAINERS

HOST MPI

- Compile with MPI inside the container, compatible with the host MPI ABI implementation
  - Use shared library and disable RPATH to compile the executables

- Replace the container MPI with the host libraries at runtime
  - Preappend host MPI library path to **`LD_LIBRARY_PATH`**

- Example: run on a HPE Cray system with SLURM and Singularity
  1. Build the container with a compatible MPI (either the same implementation or via ABI compatibility (MPICH))
  2. Preappend host MPI library path to **`LD_LIBRARY_PATH`**
  3. **`srun -n 2 singularity exec <myimage> ./myapp.x`**

     as opposite of **`singularity exec <myimage> mpirun -np 2 ./myapp.x`**, which runs the bundled container MPI

HOST MPI CAVEATS (1)

- Make sure the Container-MPI is compatible with the Host-MPI
  - Do not mix MPICH and OpenMPI
  - Note that OpenMPI is the default in most of the Linux distributions

- Host MPI library paths MUST be mounted within the container
  - Check with **`ldd`** command to see that you are linking the right libraries, e.g. Singularity with SLURM

```
srun –n 1 singularity exec <myimage> ldd ./myapp.x
```

  - Mounting host paths can introduce some conflicts with the container libraries, especially if standard paths are used (e.g. **`/lib`**, **`/var`**)
    - Make sure that glibc libraries within the container are older (and compatible) than the host's libraries, i.e.

```
ldd --version
```

HOST MPI CAVEATS (2)

- Suggestion:
  - Check if MPI works with a small test application, eg Singularity with SLURM
    - Test application (example):

```
int main( int argc, char *argv[])
{
  int myrank = -1, nranks = -1;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); MPI_Comm_size(MPI_COMM_WORLD, &nranks);
  if (myrank == 0) printf("%d\n", nranks);
  MPI_Finalize();
  return 0;
}
```

    - Compile within the container (**`test.x`**)
    - Run via: **`srun -n 2 singularity exec <myimage> ./test.x`**
    - Check if the output value is correct (2 in this case)
  - Can bundle the test with the container
- Prepare some base containers with specific MPI implementations to derive from (multistage build), eg MPICH and OpenMPI base container

GPU EXECUTION

- Can use NVIDIA docker images as base images (https://hub.docker.com/r/nvidia/cuda/)
- The driver libs are located on the host system and then bind mounted into the container at runtime
  - Can run the container on system with different versions of the NVIDIA driver
  - CUDA library installed in your container must be compatible with both drivers
    - Use a simple test application, eg checking the number of available devices
- Command line option to enable the GPU execution, e.g. **`--nv`** for Singularity
  - No input required from the user

- Recently, Singularity (v3.5) introduced support for AMD GPUs & ROCm
  - The host has a working installation of the amdgpu driver, and a compatible version of the basic ROCm libraries
  - Install ROCm libraries inside the container compatible with the host's version
  - Use the **`--rocm`** command line option

OPTIMIZED COMPILATION

- We assume that you don’t have root access on the system where you want to run the container (neither fakeroot)
  - Must build on a system where you have such access (in practice, this is usually within a virtual machine on your laptop/workstation)
- Cross-compilation when building the container for the specific target host
  - Note that some libraries apply “native” optimizations while compiling them, i.e. they apply specific optimizations related to where you build the container
- Can use dynamic dispatch, ie detect your CPU architecture (at runtime) and use the appropriate instruction set for that CPU
  - Intel MKL does that
  - Build several optimized executables and switch with an environment variable at runtime
- Can build fat binaries, ie specify multiple instruction sets and embed in a single binary

MIXING CONTAINER AND HOST LIBRARIES AND TOOLS

- A multiple steps procedure (example based on Singularity)
  - Use case: I want to use tools from the container and mix with tools/libraries from the host

1. Build a base container on your *root* machine:

```
sudo singularity build base.sif base.def
```

2. Copy the image to the host where you want to run
3. Mount the host directories of the libraries and tools within the container, e.g. vendor compilers and libraries, and open a shell:

```
singularity exec –B <mount points> base.sif /bin/bash --login
```

4. Compile your application against the host libraries and tools
5. Copy the compiled application to the *root* machine
6. Build a new container based on the previous base image and copy inside the compiled application

GENERATING OPTIMIZED RECIPE FILES

- HPC Container Maker is an open-source tool to make it easier to generate container recipe files
  - https://github.com/NVIDIA/hpc-container-maker
- Can generate Docker and Singularity recipe files from a high level Python script
- Makes it easier to create HPC applications containers by using container best practices encapsulated in building blocks
- Can easily generate specific recipe files by exploiting Python scripting, e.g.
  - Different base images
  - Different MPI implementations
  - Different optimizations

- Example: Singularity container with MPICH

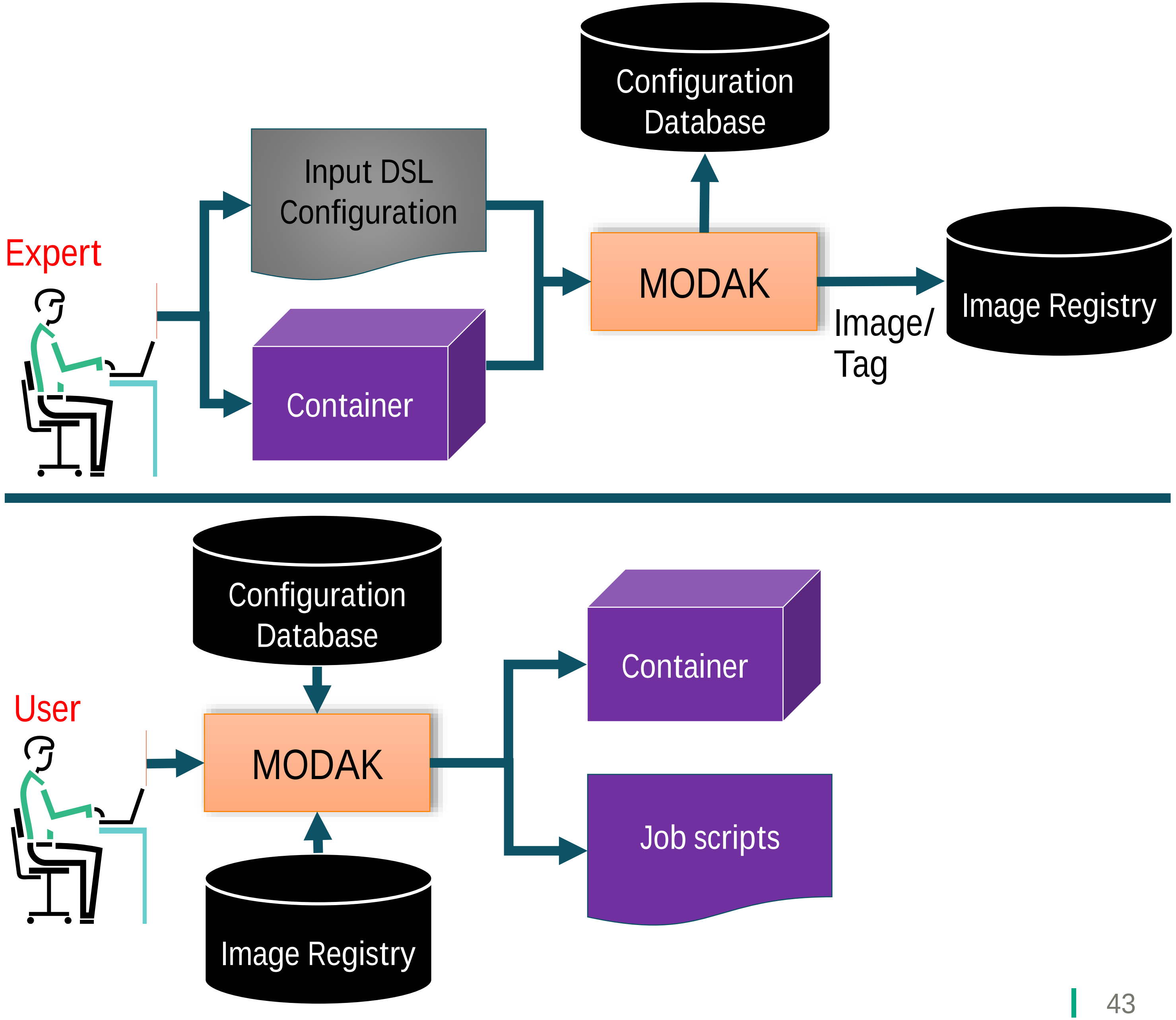
```
#!/usr/bin/env python3
import hpccm
from hpccm.building_blocks import mpich
from hpccm.primitives import baseimage

Stage0 = hpccm.Stage()
Stage0 += baseimage(image=‘debian:buster’)
Stage0 += mpich(prefix='/opt/mpich/3.3.2', version=‘3.3.2’)

hpccm.config.set_container_format('singularity’) # Choose Singularity format
print(Stage0) # Write recipe file
```

MANAGEMENT OF CONTAINERS: MODAK

- You end up with multiple versions of the container with different optimizations, stored somewhere in an image registry
  - Various combinations of compilers, MPI implementations, Linux distributions…
    - E.g. {OpenMPI, MPICH} MPI implementations ⨉ {AVX2, AV512} vector instructions ➔ 4 combinations
    - Proliferation of containers
  - How can we deploy the most appropriate container for your system?
- Introducing MODAK
  - Developed within the SODALITE EU project by us (https://github.com/SODALITE-EU/application-optimization)
  - Support HPC and Cloud systems
  - Still in a prototype phase
- Simplify the management of the containers
  - Just like Environment *Modules* for a shell, MODAK does for containers

MODAK WORKFLOW

- Expert: build the optimized containers and provide an optimization configuration DSL
  - CPU type
  - Specific libraries and configurations
  - …
- MODAK stores the configuration in a database (MYSQL), push the container in the image registry, and tag the container with an ID
- User: use MODAK to pull the specific optimized container on the system
  - Can get a batch submission script (e.g. SLURM)



CONCLUSION

CONCLUSION

- Out-of-the-box containers: portable and reproducible
  - Given the increased software complexity of emerging applications, there is a growing need for containerization within HPC
- However, to get performance you have to specialize the containers
  - Breaks portability
  - Tradeoff between portability and performance
- Presented some techniques to optimize your containers
  - No optimizations for free, need to work on the recipe files
  - Proliferation of optimized containers
- HPC Container Maker to generate the multiple recipe files
- MODAK as a solution to manage optimized containers

THANK YOU

harvey.richardson@hpe.com
alfio.lazzaro@hpe.com