

Using Singularity Containers for High Performance Computing

M. Duta¹ A. Gittings²

¹Diamond Light Source
www.diamond.ac.uk

²University of Oxford
Advanced Research Computing
www.arc.ox.ac.uk

RSE2018, September 2018

Agenda

Introduction

- Containers
- Why Singularity?

Hands-on

- Installing and testing
- Executing and sharing images
- Building images

HPC Considerations

- MPI distributed applications
- GPU processing

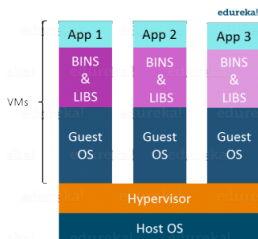
Material: <https://github.com/mcduta/RSE-2018>

- 1 Introduction
- 2 Basics: Container Execution
- 3 Advanced: Container Management
- 4 Singularity for HPC
- 5 Conclusion

Virtual Machines vs. Containers

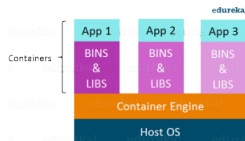
VM = virtual OS

- need: reduce TCO (hardware, maintenance)
- good: multiple OS on single system, easy recovery from failure
- bad: resource hungry, support reproducibility poorly



Container = virtual environment

- need: standardise application environment
- good: portable, lightweight, fast, reproducible
- bad: not good for everything



Containers for Science

Containers pack software apps with the environment needed for execution

- standardised software able to run on multiple platforms
- share OS kernel with other containers and processes

What can containers do for science?

- 1 escape from dependency hell (package **all** applications dependencies)
 - e.g. genomic pipelines, Python packages
 - processing pipelines can run software components on different linux distros
- 2 use/reuse and share software environments easily
 - e.g. <https://biocontainers.pro>
- 3 reproducibility
 - e.g. papers to include link to container with all software & data used

Is Docker is not enough?

- focus: micro-service virtualisation
 - develop and test in one environment
 - package application with environment for deployment into production
- philosophy: do one small thing and do it well (one process per container)
- audience: DevOps
- security: **trusted users run trusted containers**
 - users can escalate to root (FS and network access)
- strengths:
 - standardised, well documented, widely adopted
 - [Docker Hub](#) has 100k pre-built containers
- weaknesses:
 - security model good for cloud usage but bad for shared facilities
 - difficult to learn, complex model (layers)
 - reproducibility is not guaranteed

Why Singularity?

- focus: HPC, mobility of compute and reproducibility
 - once workflow defined, image can be archived to guarantee reproducibility
- philosophy: complex processing pipelines with complicated and "difficult" software dependencies packaged in single container
- audience: HPC application users
- security: **untrusted users run untrusted containers**
 - users cannot escalate to root
- strengths:
 - data-centre friendly, very easy to use
 - can run containers directly from Docker Hub
- weaknesses: less mature than Docker, smaller user community

Mobility of compute = the ability to define, create and maintain a workflow that is guaranteed to execute in a way that is independent from the different hosts, Linux distributions or service providers on which it executes.

Install and Test

Install from source as root (for full functionality).

```
VERSION=2.5.2
wget https://github.com/singularityware/singularity\
/releases/download/$VERSION/singularity-$VERSION.tar.gz
tar -xf singularity-$VERSION.tar.gz
cd singularity-$VERSION
./configure
make
sudo make install
```

Installation as root is required for Singularity security control, see singularity.lbl.gov/docs-security .

Installing from source is preferable but packages are also available.

Test and Help

Test installation (remark interoperability with Docker!)

```
singularity selftest  
singularity run shub://GodloveD/lolcow  
singularity run docker://godlovedc/lolcow
```

Obtain help

```
singularity --help  
singularity --help run  
singularity run --help
```

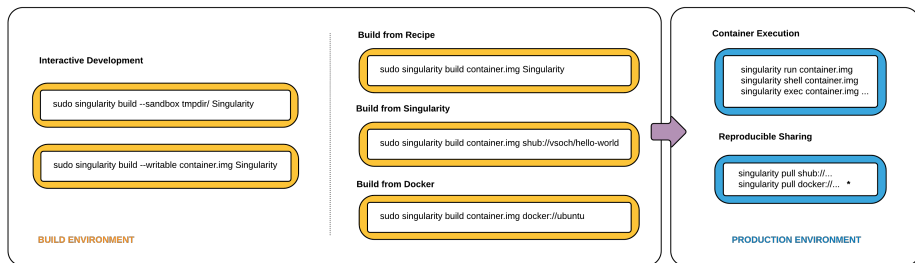
Singularity Workflow

Manage

- **modify** an existing container
- **build** a container from scratch

Use

- **execute** an existing container



* Docker construction from layers not guaranteed to replicate between pulls

- 1 Introduction
- 2 Basics: Container Execution**
- 3 Advanced: Container Management
- 4 Singularity for HPC
- 5 Conclusion

Interacting With a Container

From singularity --help

CONTAINER USAGE COMMANDS:

<code>exec</code>	Execute a <code>command</code> within container
<code>run</code>	Launch a runscrip
<code>shell</code>	Run a Bourne shell within container
<code>test</code>	Launch a testscript within container

Exercise

All commands work on all container formats.

Run `singularity run --help` and see the formats supported.

Running/Executing a Container from the Hub

exec VS. run

- run launches a **script** (predefined by the container author)
- exec executes a **command**

Exercise

Execute the command

```
singularity run shub://vsoch/hello-world
```

and observe the output. Then, run `ls`. What files are present?

Exercise

Execute the command

```
singularity exec shub://vsoch/hello-world cat /etc/*release*
```

Explain the output.

`singularity run/exec` from a Docker URI does not keep persistent images.

Pulling an Existing Image for Testing

To modify an image, a local copy is needed.

Exercise

Obtain help information from Singularity on `pull`. What does help say about Docker images and what implication that has for reproducibility?

Download a "hello world" image available from [SingularityHub](#)

```
singularity pull --name hello.simg shub://vsoch/hello-world
```

Exercise

Run `ls` to see the file or files downloaded.

Can the image be downloaded without renaming it?

Running the Image as Interactive Shell

Exercise

Obtain help on the command `shell`.

Exercise

On the pulled image, spawn a shell inside the containerised environment with singularity `shell hello.simg` and run a few commands:

```
cat /etc/*release*  
cat /proc/cpuinfo
```

Exercise

What Linux distribution is the container?
How many cores can the image "see" and why?

Exercise

Can you run a shell on a remote image, providing the URI?

Singularity and the File System

By default, Singularity tries to create a seamless user experience between container and host by bind mounting the following

- user home directory `$HOME`;
- `/tmp` and `/var/tmp`, default host devices `/dev`, `/sys`, `/proc`
- current working directory `$PWD` (if not an OS directory or already accessible via another mount)

Exercise

Run `ls` in both host OS and container shell to check contents of `/tmp`.

Exercise

Within the containerised shell, run the following

```
echo "hello, world" > $PWD/hello.txt
```

Check the file `hello.txt` was created and is visible from both the container shell and the host OS. What happens when the shell is exited, is the file permanent in the host OS file system?

Singularity and the File System

Two useful options for `shell`, `run` and `exec`:

- `--bind` mounts additional directories in the container
- `--home` specifies the home directory in the container

```
singularity shell --bind /host-path:/container-path container.simg
```

makes the directory `/host-path` outside the container appear inside the container at `/container-path`.

```
singularity shell --home /host-path:/container-path container.simg
```

makes the directory `/host-path` outside the container override the home directory within the container as `/container-path`.

run

A Singularity container can contain a "runscript", a shell script that defines the operations the container performs with the `run` command, or simply by calling the container as an executable, *e.g.*

```
singularity run hello.simg  
./hello.simg
```

The runscript is defined when the container is built (more about this later) and becomes persistent within it to a file (symlink) called `/singularity`.

Exercise

In the shell started on `hello.img`, inspect the contents of `/singularity` and understand what the runscript executes when run.

exec

exec runs commands directly from a container

```
singularity exec hello.simg cat /etc/*release*  
singularity exec hello.simg lscpu
```

Use the `--debug` flag to enable debugging and obtain more information

```
singularity --debug exec hello.simg lscpu
```

Singularity and Root Privileges

Start an interactive shell on image `hello.simg` and run the commands

```
whoami  
sudo -v
```

With `sudo -v`, a user can update the cached credentials without running a command (no terminal output means success).

Observe how Singularity cannot escalate privileges within the container started by normal user.

Exercise

Restart the shell via `sudo` and repeat the two commands above.

A user has to have root privileges outside the container to be able to use these within the container.

Singularity and Processes

Exercise

In the Singularity shell, run the `top` command.
What processes do you see?

Exercise

In the host OS, start the `firefox` browser and in the Singularity shell run the command `ps -ea | grep firefox`.
Is `firefox` that is running in the host OS visible in the container?

Exercise

What about the other way round?
Start an "infinite" command (e.g. `yes`) in Singularity shell and run `top` in the host OS. Is the process in the shell visible to the host OS?

Singularity and the Environment

A container normally inherits the host environment

```
FOO=BAR singularity exec hello.simg env | grep FOO
```

Host environment can be stripped

```
FOO=BAR singularity exec --cleanenv hello.simg env | grep FOO
```

Variables that need to be defined and immune to stripping can be specified with `SINGULARITYENV_VARNAME`, for instance

```
SINGULARITYENV_FOO=BAR singularity exec --cleanenv hello.simg env  
| grep FOO
```

The production method to initialise container specific environment variables is from the `build` recipe.

Things Learnt So Far

Docker default: share little and isolate the process, network & user space, *e.g.* user inside container cannot see hosts processes.

Singularity assumes that each application has its own container but does not seek to fully isolate containers from one another or the host OS.

Things Learnt So Far

Namespace isolation paradigm: Singularity is **transparent**

- HPC workflows do not benefit from process isolation
- run as much as possible as normal user process, change environment only enough to ensure application portability
- container can "see" (and kill!) processes on host OS and other containers
- network access is same as on host OS
- default host devices: `/dev`, `/sys`, `/proc` are bind mounted to container
- host file system: container can access any host FS and devices that any normal user process can (*inc.* optimised FS like Lustre and GPFS)

Security

- cannot escalate to root from container started as user
- root privileges within the container only if container started with root privileges on host OS

- 1 Introduction
- 2 Basics: Container Execution
- 3 Advanced: Container Management**
- 4 Singularity for HPC
- 5 Conclusion

Writable Shells

Using pre-defined images only is limiting, what about making changes?

Limited changes can be made via the `shell --writable` option.

Exercise

Obtain help information for the `shell` command on writable containers.
Try the command `sudo singularity shell --writable hello.simg`

The image pulled earlier is a read-only **squashfs** file, which needs to be changed to writable **ext3** format using the `build` command (more about it later)

```
sudo singularity build --writable hello.img hello.simg
```

This can be achieved also using the original URI

```
sudo singularity build --writable hello.img  
shub://vsoch/hello-world
```

Writable Shells

Suppose we need to install a package. For instance

```
singularity shell hello.img pstree
```

ends with a shell error as the package providing that command is missing.

Exercise

Run a shell via `sudo` on the writable image and install the `psmisc` package:

```
sudo singularity shell --writable hello.img  
apt-get install psmisc
```

Exercise

Still within the shell, check the command `ps-tree` can now be executed. Exit the Singularity shell. Repeat the `exec` run of `ps-tree` in order to check that the change to the container is permanent.

Writable Shells

Caution

The option `--writable` is crucial; without it, changes are still possible but are not permanent – once the shell exited, changes are lost.

Limitation

The `--writable` shell option predefines a fixed amount of (buffer) disk space in addition to the existing image size, which limits the size of the changes to a writable shell.

Lesson: using a writable shell to modify an existing image is possible but is a limiting option for production purposes.

Management Commands

What options exist for managing a production container?

From singularity `--help`

CONTAINER MANAGEMENT COMMANDS:

<code>apps</code>	List available apps within a container
<code>bootstrap</code>	<i>*Deprecated*</i> use <code>build</code> instead
<code>build</code>	Build a new Singularity container
<code>check</code>	Perform container lint checks
<code>inspect</code>	Display containers metadata
<code>mount</code>	Mount a Singularity container image
<code>pull</code>	Pull a Singularity/Docker container to <code>\$PWD</code>

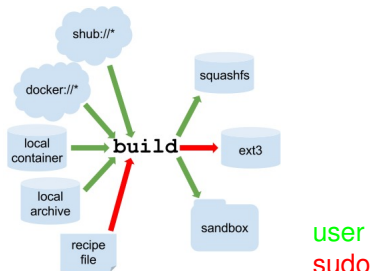
This workshop focuses on `build` – the main tool for container management.

The rest of the commands are only briefly introduced.

build

The "Swiss Army knife" for container management

- builds new images from other images
- builds new images from scratch using a recipe
- converts existing images between 3 major container formats



- ext3 – the default for many popular Linux distros
- squashfs – compressed read-only file system for Linux
- sandbox – a chroot directory for interactive development

Building a Simple Container

Consider the task is to build a "lolcow" pipeline, similar to `shub://GodloveD/lolcow` (Ubuntu).

Exercise

Run Singularity on the "lolcow" URI above to see what functionality to expect.

The task is

- implement the runscript `fortune | cowsay | lolcat`
 - `fortune` echoes quote from database
 - `cowsay` echoes cow that says something (from argument)
 - `lolcat` rainbow colours output
- bootstrap from a Docker Centos v. 7 image
- add options for
 - testing
 - optional command line arguments

Building a Simple Container

Three stages

- build a sandbox to understand what needs to be done
- build an image from a basic recipe
- improve the recipe

Sandbox build

Start with a sandbox build to check what needs to be installed and how.

First, build the sandbox from the latest Centos image from Docker

```
sudo singularity build --sandbox centos/ docker://centos:latest
```

build converts the Docker image to the latest Singularity image format.

Start a writable shell on the sandbox

```
sudo singularity shell --writable centos/
```

No disk space limitations any longer!

Unlike a writable image, a writable sandbox is not limited by (buffer) disk space.

Sandbox build

Exercise

In the host OS, run `ls` on the `centos` directory.
In the Singularity shell, run `ls` on `/home/workshops`.
Is what you observe expected?

Recall

`chroot` is an operation that changes the apparent root directory for the current running process (and its children).

Exercise

In the writable shell, try the command `fortune`.

Sandbox build

Exercise

Make sure the Extra Packages for Enterprise Linux (EPEL) are available:

```
yum -y install epelrelease
```

Exercise

Check what package provides `fortune` by doing

```
yum whatprovides fortune
```

Exercise

Install this package with

```
yum -y install fortune-mod.
```

Exercise

Verify the command `fortune` works.

Sandbox build

The complete set of commands for the "lolcow" pipeline is

```
yum -y install epel-release
yum -y install fortune-mod
yum -y install cowsay
yum -y install wget
yum -y install ruby
yum -y install tar
yum -y install gzip
wget https://github.com/busyloop/lolcat/archive/v99.9.11.tar.gz
tar -xf v99.9.11.tar.gz
cd lolcat-99.9.11/bin
gem install lolcat
```

The above is a mixture of package installations and installation from source, more complicated than in Ubuntu, but useful for illustrating sandboxing.

Sandbox build

Exercise

Copy and paste the above set of commands in the sandbox shell.
Run the "lolcow" pipe `fortune | cowsay | lolcat` to verify installation.

Exercise

Exit the shell. Restart the shell with `singularity shell centos/`. Rerun "lolcow". Are the sandbox changes permanent?

Exercise

Exit the shell. Run `fortune` via the Singularity `exec` option with `singularity exec centos/ fortune`. Does it work?

Exercise

How about `exec` on the whole "lolcow", i.e. `singularity exec centos/ fortune | cowsay | lolcat`? How do you explain this?

Sandbox build

The sandbox can be converted into a `squashfs` Singularity image (immutable, hence a good format choice for reproducibility)

```
singularity build centos.simg centos/
```

Exercise

Does the operation above require `sudo`?

Exercise

Run `singularity exec` on the new image and verify `fortune` can execute. Start a shell on the new image and run the entire "lolcow" pipe.

Recipe build

A method for creating the "lolcow" pipe in Centos now exists.

This method is cast into a "recipe" for building the container automatically, *e.g.*

```
sudo singularity build container.simg container.def
```

A recipe (definition file)

- enhances reproducibility by automating all above manual operations
 - specifies the base OS
 - defines software installations
- adds functionality
 - the ability to run a default script
 - the ability to run tests

This is the recommended method to generate production Singularity containers.

Docker is very similar in the use of "Dockerfiles".

Recipe build

Exercise

Open the file `centos-lolcow.def` with the `gedit` editor.
Notice the recipe parts marked by `%post` and `%runscript`.

Exercise

Build a container from this recipe with the command
`sudo singularity build centos-lolcow.simg centos-lolcow.def`

Exercise

Run the container with either of the commands
`singularity run centos-lolcow.simg`
`./centos-lolcow.simg`

Exercise

Run the container with the command
`singularity exec centos-lolcow.simg cat /singularity`
What does the output represent?

build Recipe Structure

- ❶ **header** specifies the base OS and options to use to build the container
 - Bootstrap references the base (*e.g.* docker, shub)
 - From names the container (shub) or reference to layers (docker)
- ❷ **sections** add content or execute commands during the build process
 - %help
 - %setup commands run **outside container** (host OS) after base OS installed
 - %files copies files from host OS into container
 - %labels container metadata
 - %environment defines environment vars for runtime
 - %post commands run **within container** after base OS installed
 - %runscript defines scriptlet persistent within container (executed via singularity run or by running container directly)
 - %test defines test scriptlet (executed at end of build and via singularity test)

%post and %runscript are the most widely used sections.

Improved Recipe build

To produce an image more akin to an application, improve the recipe by adding

- help
- labels
- test
- extended runscript that can take arguments

Exercise

Open the file `centos-lolcow-args.def` with the `gedit` editor. Notice the additions to `centos-lolcow.def` that implement the above. In particular, pay attention to `%test` and `%runscript`.

Exercise

Build a container from this recipe with the command
`sudo singularity build centos-lolcow-args.simg centos-lolcow-args.def`

Improved Recipe build

Exercise

Run `./centos-lolcow-args.simg` to obtain application help.
Create a file `inp.txt` containing some text. Run the container with
`./centos-lolcow-args.simg -i inp.txt`

Exercise

Now run the container with both input and output arguments
`./centos-lolcow-args.simg -i inp.txt -o out.txt`
Run `cat` on file `out.txt` to observe the output.

Improved Recipe build

Exercise

Run the following commands and explain the observed output

```
singularity help centos-lolcow-args.simg  
singularity test centos-lolcow-args.simg
```

Exercise

Explore the inspect option with

```
singularity inspect centos-lolcow-args.simg  
singularity inspect --runscript centos-lolcow-args.img  
singularity inspect --test centos-lolcow-args.img  
singularity inspect --environment centos-lolcow-args.img
```

Advanced build Options

- single container with two or more different apps
 - each have own runscripts and custom environments but
 - can be useful for apps have (almost) equivalent dependencies
 - achieved via sections such as %apphelp, %apprun, etc.
- extra build arguments
 - `--force` – forces new base OS be bootstrapped into existing container
 - `--section` – build a single section of the recipe file, *e.g.* having edited the `%environment` section, the image can be rebuilt only for that section (avoiding building the rest) with `sudo singularity build --section environment image.simg recipe`. (*N.B.* The recipe file is saved into container's meta-data, using the `--section` option may render this meta-data useless, which not good for reproducibility.)
 - `--notest` – skip tests
 - `--checks` – define security checks to be run on demand

Best Practices for build Recipes

- build containers from recipe (instead of a sandbox)
 - better reproducibility
 - mitigates "black box" effects
- global installs specified in `%post` should go into operating system locations (e.g. not `/home`, `/tmp`, or any other directories that might get commonly binded on)
- provide help via the `%help` and `%apphelp` sections
- make container internally modular via apps, with shared dependencies under `%post`
- files installed in container should never be owned by actual users but by a system account (UID less than 500)
- any special environment variables to be defined, are to be added to `%environment` and `%appenv` sections of recipe
- ensure the container files `/etc/passwd`, `/etc/group` and other sensitive files have nothing but the bare essentials

Things Learnt Updated

- Singularity assumes that users have access to
 - a build system where user **can** become root and
 - a production system where user **cannot** escalate privileges.
- Singularity workflow
 - develop container on own desktop (using `sudo`)
 - remote copy container to an HPC resource
 - run on the HPC resource as normal user (without `sudo` access)
- Singularity is good friends with Docker
 - no need for Docker to be installed
 - Docker images can be run/built as Singularity images
 - Docker images can be pulled
 - Docker bases and layers can be assembled into Singularity images

- 1 Introduction
- 2 Basics: Container Execution
- 3 Advanced: Container Management
- 4 Singularity for HPC**
- 5 Conclusion

High Performance Computing

The selling line: Singularity was designed to run containers on HPC resources.

Essential HPC features

- **parallel execution** (multi-threaded and/or distributed)
- **accelerators** (GPUs)
- **fast disk I/O** (large data)

Desirable HPC features

- good integration with resource managers

We are discussing MPI and GPUs.

Message Passing Interface

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv){
    int size, rank;

    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);

    printf ("hello from process %d of %d\n", rank, size);

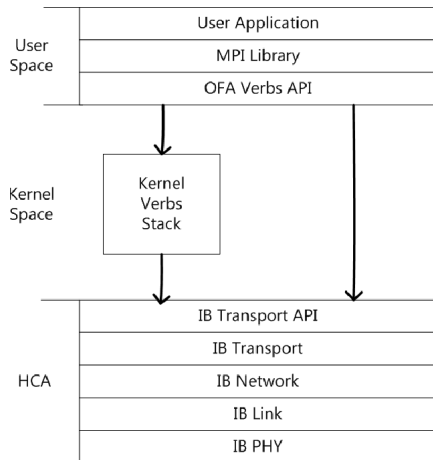
    MPI_Finalize ();
}
```

```
$ mpicc hello.c -o hello
$ mpirun -np 4 -hostfile hosts.txt ./hello
hello from process 2 of 4
hello from process 1 of 4
hello from process 3 of 4
hello from process 0 of 4
```

The MPI Standard for Distributed Computing

- syntax and semantics of a core of library routines designed for writing portable message-passing programs (in C and Fortran)
- programming model is independent processes running concurrently
 - each process executes one single program
 - each process has its own memory address space
 - processes communicate messages (data) where needed
- appropriate abstraction ensures functionality and portability across
 - different implementations of the MPI library
 - different platforms without modifications

MPI Performance = Implementation + Hardware



Keywords:

- **InfiniBand (IB)** = HPC network communications standard
- **OFA** = OpenFabrics Alliance
- **OFED** = OFA IB software stack
 - user/kernel level protocols/APIs
 - `ibverbs` = library that allows userspace processes to use InfiniBand/RDMA "verbs" directly (hardware/OS abstraction layer)

MPI and Singularity

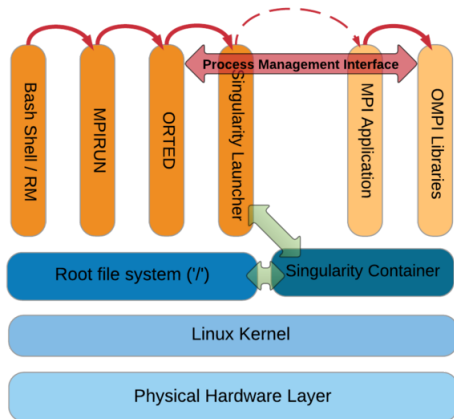
MPI components are present both inside and outside the container

- `host`
 - executes the `mpirun` launcher and controls the MPI host list
 - controls messaging over network (between hosts) and within a host
- `container`
 - containerised application and MPI libraries
 - containerised OFED components for appropriate IB support

Remarks:

- MPI software outside must be newer or equal to the container version
- variations to the above are possible by mount binding paths to MPI software installed on host

Out-of-the-Box Compatibility with OpenMPI



- `mpirun` called by resource manager (or directly from shell)
- OpenMPI calls the process management daemon (ORTED)
- ORTED launches container requested by `mpirun`
- Singularity builds container and namespace environment
- Singularity launches the MPI application within the container
- MPI application launches and loads the OpenMPI libraries
- OpenMPI libraries connect back to ORTED via the Process Management Interface (PMI)
- application processes run within container as they would on host

Basic Container with MPI Support

Exercise

Edit file `ubuntu-mpi-hello.def` and understand the recipe:

- start with a Docker image for Ubuntu 16.04 (LTS)
- build the OpenMPI library from source and
- use the OpenMPI compiler wrappers to build two simple applications (copied to container from `./src`)

The MPI "hello-world" image is built with

```
sudo singularity build ubuntu-mpi-hello.simg ubuntu-mpi-hello.def
```

Basic Container with MPI Support

```
$ mpirun -np 4 singularity exec ubuntu-mpi-hello.simg hello
hello from process 0 of 4
hello from process 1 of 4
hello from process 2 of 4
hello from process 3 of 4
```

```
$ mpirun -np 4 singularity exec ubuntu-mpi-hello.simg ring 1000
```

```
----- before communication
```

```
process 0: mean of data sent to 1 = 4.865778e-01
process 1: mean of data sent to 2 = 5.122776e-01
process 2: mean of data sent to 3 = 5.041534e-01
process 3: mean of data sent to 0 = 4.846409e-01
```

```
----- after communication
```

```
process 0: mean of data received = 4.846409e-01
process 1: mean of data received = 4.865778e-01
process 2: mean of data received = 5.122776e-01
process 3: mean of data received = 5.041534e-01
```

```
correct communication
```

```
communication time = 4.019204e-05
```


Basic Container with MPI Support

Simple!

The Singularity and application combo works in a very familiar fashion:

- `mpirun` call with usual parameters
- command line args to application passed on normally

What about performance?

- basic MPI container recipe has no explicit support for IB
- enter the Ohio State University (OSU) set of micro-benchmarks

Basic Recipe for OSU Benchmarks

Exercise

Edit file `ubuntu-osu-benchmarks.def` and examine the recipe

- the OpenMPI library is installed and
- the OpenMPI compiler wrappers are used to build the OSU micro-benchmarks
- pattern similar to the "hello-world" recipe (no containerised IB support)

Basic Recipe for OSU Benchmarks

Compare a native run of the OSU bandwidth test with a container run

- p2p communication using 2 processes
- bandwidth measured across the Mellanox FDR IB network

Native

```
$ mpirun -np 2 osu_bw
# OSU MPI Bandwidth Test v5.4.0
# Size      Bandwidth (MB/s)
...
4096        3050.72
8192        4701.28
16384       4925.92
32768       5414.88
65536       5491.43
131072      5502.49
262144      5537.50
524288      5577.15
1048576     5669.64
2097152     5576.75
4194304     5704.30
```

Singularity

```
$ mpirun -np 2 singularity exec ubuntu-osu-benchmarks.simg osu_bw
libibverbs: Warning: couldn't open config directory '/etc/libibverbs.d/'.
...
[[32751,1],1]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:
...
Another transport will be used instead, although this may result in
lower performance.
...
-----
# OSU MPI Bandwidth Test v5.4.1
# Size      Bandwidth (MB/s)
...
4096        552.35
8192        804.75
16384       1006.86
32768       1087.11
65536       121.40
131072      134.64
262144      135.48
524288      135.75
```

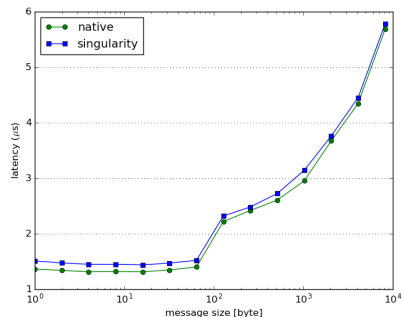
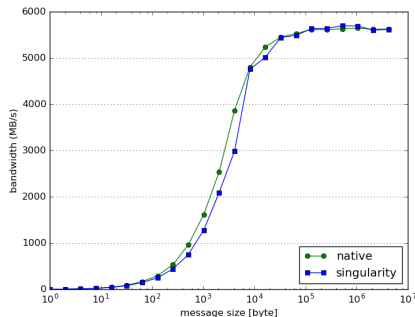
Full Recipe for OSU Benchmarks

Exercise

Edit file `ubuntu-osu-benchmarks-ibverbs.def` and remark that the same recipe for building OpenMPI and the benchmarks as in `ubuntu-osu-benchmarks.def` are now preceded by

- the installation of library for InfiniBand support and
- the addition of the appropriate Linux kernel modules

OSU Benchmarks

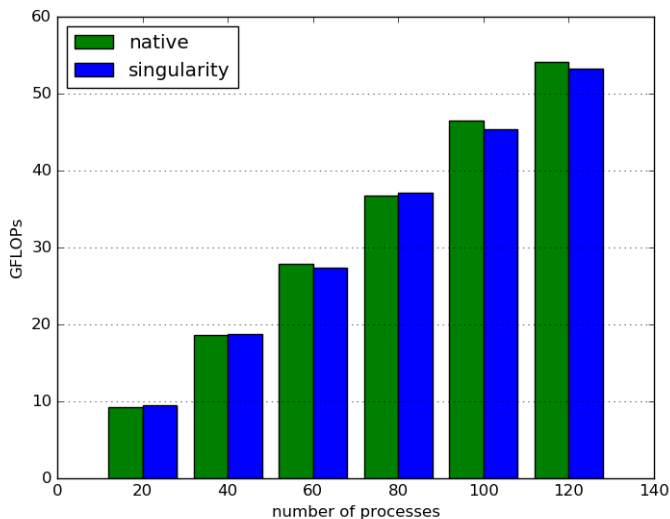


A production recipe!

- full bandwidth and small latency penalty
- good performance on **all** OSU benchmarks (inc. collective comm.)

HPCG Benchmarks

The recipe gives (nearly) native performance on other MPI applications.



Singularity for MPI Applications

General recipe presented

- container library support for InfiniBand (OFED)
- matched by OS IB support (OFED + drivers) and IB hardware on the host
- covers 90% of what most users need

An improved recipe adds to this

- fine tuning by vendor specific network support in container
- example: <https://community.mellanox.com/docs/DOC-3148>
 - MLNX_OFED (Mellanox optimised OFED distribution)
 - HPC-X Toolkit (drivers, optimised engines for hardware offload, ...)

Singularity and CUDA

Old Singularity (v.<2.3)

- perceived portability: CUDA toolkit and Nvidia drivers installed in container
- reality: such containers work on host systems that match the containerised Nvidia version (host updates problematic)

New Singularity (v.>2.4)

- native support for GPUs with the `--nv` option
- only CUDA has to be installed in the container
- genuine portability
 - Nvidia driver libraries on the host are bind mounted into container at runtime
 - one container can work on different systems with different driver versions
 - this assumes containerised CUDA is compatible with host driver
 - good news: Nvidia is committed to maintain backward compatibility

Singularity and CUDA

Exercise

Edit `ubuntu-gpu-basics.def` and understand its function.

Without `--nv`, this container can break showing driver mismatch:

```
$ singularity run cuda.simg
Failed to initialize NVML: Driver/library version mismatch
```

With `--nv`, observe the difference between CUDA (`nvidia-smi`) and host driver versions

```
# run container
$ singularity run --nv cuda.simg
+-----+
| NVIDIA-SMI 396.44                Driver Version: 384.125                |
|-----+-----+-----+
# run native
$ nvidia-smi
+-----+
| NVIDIA-SMI 384.125                Driver Version: 384.125                |
|-----+-----+-----+
```

Singularity and Resource Managers

Singularity can run containers as if they were normal applications on the host

- integration with schedulers is straightforward
- past integration support with Slurm is deprecated

Cluster can pose specific problems, though

- multiple node runs must make use of scheduler integration with host MPI
- application command line arguments
 - passed transparently from Singularity command line
 - dealt with by a runscript (similar to "lol-cow" with args)
- working directories on a file system shared across nodes
 - \$PWD already mounted on OS, cannot be bind mounted in container
 - solution is to make use of `--home $PWD`

Cluster admin guide: singularity.lbl.gov/admin-guide .

- 1 Introduction
- 2 Basics: Container Execution
- 3 Advanced: Container Management
- 4 Singularity for HPC
- 5 Conclusion**

Singularity on HPC Resources is (Relatively) Easy

- On a system with sudo access, the user can
 - built and test container on workstation from recipe
 - adapt existing container from Singularity/Docker hub
- Container is run on cluster without privileged access
 - all the benefits of HPC with small overheads
 - MPI support is relatively easy to set up

Thank you!