Running Scientific Applications on HPC Infrastructure Using Singularity: A Case Study

Jeremy Cohen
Imperial College London

18th July 2018

Introduction

- Singularity[1] is a container platform that is gaining traction in the scientific community, particularly for use in HPC environments
- `https://www.sylabs.io/`
- I have no connection with the Singularity team
- This presentation gives my take on Singularity from the perspective of an end-user discovering and learning how to use the tool

[1] Kurtzer GM, Sochat V, Bauer MW (2017) Singularity: Scientific containers for mobility of compute. PLoS ONE 12(5): e0177459. https://doi.org/10.1371/journal.pone.0177459

Introduction

- In this talk I will:
  - Provide some background and motivation on my use of Singularity
  - Give a brief overview of the software providing the use case – the Nektar++ spectral/hp element framework
  - Explain how to build a singularity container in a standard Linux desktop/server environment
  - Show how to use this container to run parallel MPI jobs in an HPC cluster environment

Introduction

- Containers enable packaging of a complete software environment, base OS, libraries, applications, etc.
- Simplified transfer/deployment of applications/app. stacks
- Container-type platforms have been around for some time with systems like Solaris Zones, Linux Containers, etc.
- Docker (`www.docker.com`) is probably the best known recent container platform
- Docker provides many features in addition to the underlying container technology offering simplified container access, management, deployment and a range of other capabilities.

Introduction

- Docker is generally not available on HPC clusters
- Sysadmins reluctant to offer it due to potential security issues

  This is ultimately a result of docker's architecture with a privileged daemon and the need to be able to interact with the daemon to run/manage containers
- Singularity provides a different approach running as a user-space application rather than using a daemon
- This makes Singularity better placed for use on HPC platforms
- Many discussions/explanations about this online, e.g. [2]

[2] https://www.reddit.com/r/docker/comments/7y2yp2/why_is_singularity_used_as_opposed_to_docker_in/

Motivation

- Fair bit of experience working with Docker/containers
- Working with the Nektar++ spectral/hp element framework, including undertaking parallel MPI runs on HPC infrastructure
- Using Nektar++ on the local HPC cluster either requires
  - a time-consuming and sometimes challenging build of the software by each user in their local user space, or
  - support from the sysadmins in packaging each new release
- Containers seem ideal here...but Docker not available
- Since the HPC cluster has Singularity available, I set out to try and use Singularity to simplify undertaking parallel runs of Nektar++ on the cluster

Nektar++

- Nektar++ (http://www.nektar.info) is a spectral/hp element framework for undertaking high-order simulation of fluid and air flow problems across 2 and 3-dimensional meshes
- Use cases in a wide range of scientific, engineering and medical fields including automotive and aeronautical engineering and cardiac electrophysiology

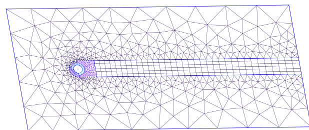


Figure: Mesh of a 2D cylindrical obstruction in a flow. Generated using Gmsh (http://gmsh.info)

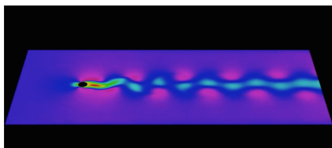


Figure: Visualising simulation of the flow of an incompressible fluid around the obstruction using Nektar++'s incompressible Navier-Stokes solver

Nektar++

- Nektar++ is an advanced C++ code with many dependencies
- Some dependencies optional and can improve performance in specific calculations
- Mature CMake-based build system that has improved greatly over the last few years
- Many third-party dependencies can be automatically built by build system
- Building on an HPC platform can, however, be time consuming and challenging - different compilers, versions of dependencies, etc.

Building a Singularity container

- Singularity can now use Docker containers but I opted to build my own container using Singularity
- Containers can be started by a non-root user without any special permissions or group membership
- When in a container shell or running commands in a container, you retain your current user details from the host system
- I investigated different approaches to building a container and opted to go for a writable sandbox directory container

Building a Singularity container

- `https://singularity.lbl.gov/docs-build-container` provides a good overview of the different approaches to building a container
- For the initial container build process requiring many stages that will need to be determined as things progress, a writeable sandbox is a practical option
- Using Singularity's Ubuntu base image (14.04) to start from (but could equally use a docker Ubuntu image...)

Building a Singularity container: Getting started

- We could simply pull the singularity container:

```
singularity pull --name nektar.img shub://singularityhub/ubuntu
```

- ...and then open a shell in the container:

```
singularity shell [--writeable] nektar.img
```

- BUT this gives us non-root, read-only access to the downloaded container
- In order to start building our new container, we instead create a writable sandbox directory from the base container:

```
sudo singularity build --sandbox nektar-singularity/
shub://singularityhub/ubuntu
```

Installing/configuring software

- Open a shell in your container, specifying the writable flag to ensure that the contents of the container can be changed:

```
sudo singularity shell --writable nektar-singularity
```

- Depending on the configuration of your Singularity deployment, various directories from your host system may be bound to and visible in the container shell (see `http://singularity.lbl.gov/docs-mount`)

- You should, for example, have access to your home directory from the host system within your container
  NOTE: this will be the root user's home directory if you've used sudo to access the container

Installing/configuring software

- Now we need to undertake any installation/configuration of software within the container
- We will also create a directory that we can use to place output data into during our job runs (more on this later...):

```
mkdir /data
chmod 777 /data
```

- Ultimately we should use a `recipe` file (`http://singularity.lbl.gov/docs-recipes`)
  - will help provide a reproducible, more manageable build that can be more easily automated, e.g. as part of a build system
  - could then run: `sudo singularity build nektar.simg nektar.recipe`

Simplified Nektar++ Singularity recipe file example

```
Bootstrap: shub
From: singularityhub/ubuntu

%help
    This container includes Nektar++ 4.5.0 build against MPICH.

%labels
    Maintainer jcohen
    Version v0.6

%post
apt-get update && apt-get dist-upgrade
apt-get install -y git-core mpich libmpich-dev build-essential cmake flex bison liblapack-dev libz-dev
mkdir -p /usr/src && cd /usr/src
git clone https://gitlab.nektar.info/nektar/nektar.git && cd nektar
mkdir build && cd build
cmake -DNEKTAR_USE_MPI:BOOL=ON -DCMAKE_INSTALL_PREFIX="/usr/local/nektar" ..
make
make install
mkdir /data
chmod 777 /data
```

Working with MPI codes

- If you're building/installing a parallel MPI code, there are various considerations to make
- It's important to understand how Singularity operates in parallel environments with MPI
- Some useful information is provided by Singularity at: `http://singularity.lbl.gov/docs-hpc#integration-with-mpi`
- As part of a proposal on container versioning, `https://github.com/open-mpi/ompi/wiki/Container-Versioning` provides a nice overview of different MPI deployment scenarios for containers in an OpenMPI context

Working with MPI codes

- Singularity has integrated support within OpenMPI (2.1.x+?)
- Nonetheless, other MPI versions can be used
- My target platform provides Intel MPI, given ABI Compatibility (`https://www.mpich.org/abi/`) I should then be able to build my code using MPICH but still run it on a platform that offers Intel MPI
- This is the approach I have tested and the reason for building my target application with MPICH

Bundling your container

- Once all the software is installed in your sandbox container directory, you are ready to package it for deployment
- Ensure permissions are correct for non-root user access
- Make any other final configuration changes required
- Exit the container and then convert it into a read-only Singularity *squashfs* container image file:

```
sudo singularity build nektar.simg nektar-singularity/
```

Container deployment

- You can now deploy your container image to your target platform(s)
- You can publish your image to Singularity Hub (`https://singularity-hub.org/`)
- In my case I only wanted to run my test container on local HPC infrastructure so it was placed on a local server for download via HTTP(S)

Imperial College Research Computing Service

- Using HPC infrastructure provided by the Imperial College Research Computing Service (http://doi.org/10.14469/hpc/2232)
- Group of systems available to Imperial researchers/academics
- I'm working with the CX1 general purpose compute cluster which is accessed via the PBS job scheduler

Running Nektar++ sequentially

- We can simply open a shell in our container (on the node where we built it) and run a Nektar++ solver from there
- Starting with the 2D cylinder mesh shown in slide 7, in the file `Cyl.xml`

```
singularity shell ./nektar.simg
> /usr/local/nektar/bin/IncNavierStokesSolver ./Cyl.xml
```

- Alternatively, you can run the solver directly from inside the container using `singularity exec`:

```
singularity exec nektar.simg \
/usr/local/nektar/bin/IncNavierStokesSolver ./Cyl.xml
```

Running in parallel with MPI

- Setting things up for undertaking parallel runs of the code using MPI
- Target platform is one of Imperial's HPC clusters:
  - uses PBS for job submission
  - provides Singularity as a system module
  - provides Intel MPI runtime that we'll need to run our simulations
- Our executables within the container are linked against the MPICH libraries within the container
- These libraries will be available to the system when running the target executable within the container context

Running in parallel with MPI

- The description at `https://github.com/open-mpi/ompi/wiki/Container-Versioning#scenario-1-containers-for-application-only` highlights the general setup being used here
- In order to run our simulation via Singularity, we now need:
  - A copy of our singularity image, accessible to all compute nodes in our cluster
  - Our input data file(s)
  - A PBS script that will initiate the job run
- The Singularity image is pulled via HTTP(S) and stored to a location on the cluster where compute nodes can access it

Run configuration: Binding the output directory

NOTE: The target cluster being used here provides a local temporary directory on each compute node (not a shared directory) which should be used for input/output for running jobs. This significantly assists with performance over use of a shared file store. This directory will be referred to as `$TMPDIR`

- `$TMPDIR` is not within our home directory (it is on local disk space on each compute node)
- When we run our container and are in the container context, we are unable to see this directory by default

Run configuration: Binding the output directory

- We therefore need to bind `$TMPDIR` into a directory in our container - we'll use `/data` that we created earlier
- The directory is bound by specifying the `-B` switch when running the singularity executable, e.g.

```
-B $TMPDIR:/data
```

- Any files placed in `$TMPDIR` on the host will now be visible in `/data` in the container (and vice versa)

Running on the cluster

We'll now look at some examples of running jobs on our target cluster. We're using the following files:

- Singularity container file: `netkar.simg`
- Nektar++ solver: `/usr/local/nektar/bin/IncNavierStokesSolver`
- Input data file: `Cyl.xml` (2D flow around cylinder example)
- `module load singularity` must be run on the cluster to access singularity

Example 1: Running sequentially on a compute node - show baseline performance (in interactive session)

- Copy `Cyl.xml` input file to `$TMPDIR`

Running on the cluster

- Run:

```
singularity exec -B $TMPDIR:/data \
$WORK/singularity/nektar.simg \
/usr/local/nektar/bin/IncNavierStokesSolver /data/Cyl.xml
```

- Some example output from the sequential computation:

```
Writing:  "/data/Cyl_0.chk" (0.0110159s, XML)
Writing:  "/data/Cyl_1.chk" (0.0219893s, XML)
Writing:  "/data/Cyl_2.chk" (0.0221715s, XML)
Writing:  "/data/Cyl_3.chk" (0.0218782s, XML)
Writing:  "/data/Cyl_4.chk" (0.0218389s, XML)
```

Running on the cluster

Example 2: Running in parallel on a single compute node (in interactive session)

- Copy `Cyl.xml` input file to `$TMPDIR`
- Run:

```
mpiexec singularity exec -B $TMPDIR:/data \
$WORK/singularity/nektar.simg \
/usr/local/nektar/bin/IncNavierStokesSolver /data/Cyl.xml
```

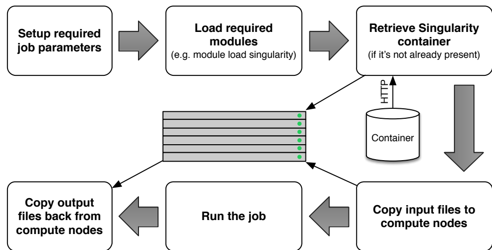
- Some example output from the sequential computation:

```
Writing:  "/data/Cyl_0.chk" (0.00601912s, XML)
Writing:  "/data/Cyl_1.chk" (0.00399995s, XML)
Writing:  "/data/Cyl_2.chk" (0.00594401s, XML)
Writing:  "/data/Cyl_3.chk" (0.00383973s, XML)
```

Running on the cluster

Example 3: Running in parallel across multiple cluster compute nodes)

- Here we prepare a full job submission with a qsub script that we can submit to the PBS scheduler.
- Our script needs to carry out the following tasks:



Using Singularity in an HPC environment

Some key things to note about how singularity is used in a cluster environment:

- `mpirun`/`mpiexec` is used to call the singularity executable
- We are using the host system's MPI executable/daemon to spawn the singularity processes
- The MPI libraries within the container, which are loaded when our executable runs, are communicating with the MPI daemon on the host system [`http://singularity.lbl.gov/docs-hpc#integration-with-mpi`]
- Without a Singularity-aware MPI implementation, MPI is opening the container for every process. A Singularity-aware MPI implementation (currently OpenMPI 2.1.x+) can reduce overhead by running processes from only one instance of the container for each physical node [`https://groups.google.com/a/lbl.gov/d/msg/singularity/I9v5-14A8W8/b5FcuuJzKgAJ`]

Conclusions I

- Demonstrated a basic example of building and using a Singularity container for an HPC code
- Provides a great way for simplifying the building and running of codes in an HPC environment
- Singularity's compatibility with Docker containers is a further benefit
- Possible questions around performance (e.g. when building against MPI libraries on one system to run on another)
- No concrete performance tests undertaken yet, may not be a significant issue

Conclusions II

- Taking advantage of specialist networking hardware/interconnects more complex when building on separate external platform
- Going forward, I'm keen to undertake some performance tests to understand more about some of these points
- If you have access to an HPC cluster that provides Singularity, it can offer a great way to simplify your HPC workflow and the deployment of new versions of your code

Thank you

Questions?

jeremy.cohen@imperial.ac.uk

Acknowledgements:

- The Imperial College Research Computing Service: `http://doi.org/10.14469/hpc/2232`
- Chris Cantwell and the Nektar++ team (`http://www.nektar.info`)
- JC acknowledges support from EPSRC under RSE Fellowship grant EP/R025460/1