Hiding large numbers of files in container overlays

ALEX RAZOUMOV
alex.razoumov@westdri.ca

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants

- Email training@westdri.ca
- Our winter/spring training schedule `https://bit.ly/wg2023a`
  - biweekly webinars
  - weekly online courses starting in February
  - in-person workshops @SFU @UBC
  - in-person spring/summer schools @UVic @SFU

Many files on a distributed filesystem (DFS) such as Lustre

- Many of our cluster filesystems are Lustre-based
  - https://docs.alliancecan.ca/wiki/Storage_and_file_management#Filesystem_quotas_and_policies
  - highly scalable: up to hundreds of servers, tens of PBs, over a TB/s of aggregate throughput
  - employ multiple metadata + object storage servers that manage access to blocks stored on thousands of drives
  - multiple clients (individual cluster nodes) can access the DFS through the network
  - all clients see a unified namespace for all of the files and data in the filesystem
  - multiple levels of cache + data replication to improve I/O speeds and keep data highly available

✔ One of the design goals: reach high bandwidth for large I/O operations (= reading/writing few large, unbroken data streams)

✘ Having a large number of small files can lead to poor performance and strain the metadata server(s)

⇒ for best performance minimize the number of open/close operations on the filesystem

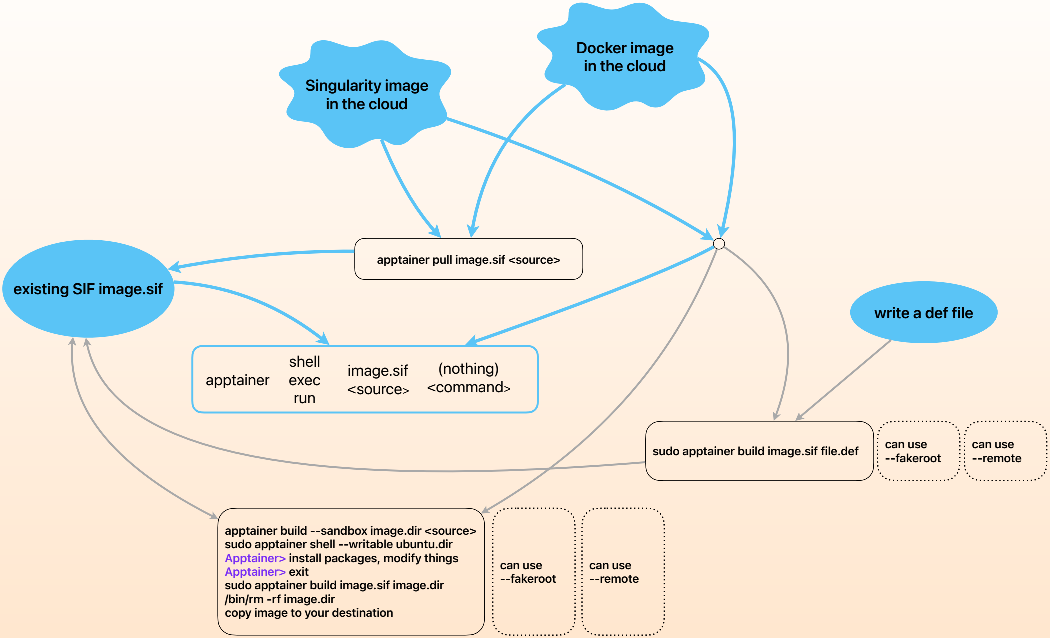
⇒ in other words, avoid high IOPS workflows (small reads/writes, writing many small files)

Possible solutions

- If you already have many small files:
  - use archival tools such as tar, DAR to pack them into few archive files
  - `https://docs.alliancecan.ca/wiki/Handling_large_collections_of_files`
  - `https://docs.alliancecan.ca/wiki/Dar`
  - only a partial solution ...
- Modify your code to streamline output to fewer, larger files
- Use local (to the node) temporary storage: `$SLURM_TMPDIR` + copy out at the end
- Use an SQL database inside your code
  - does not apply to most workflows?
  - upcoming Feb-28 webinar
- ✔ Use container overlays
  - an overlay image is a file formatted as a filesystem
  - a container sees files inside, while the host OS sees only one large overlay image file
  - if your code does lots of small reads/writes, these are cached by the OS and Apptainer software before reaching the host's filesystem
- And few others ...

Singularity / Apptainer containers

- Create a custom, secure virtual Linux environment (a container) that is different from the host Linux system, uses
  - uses kernel namespaces to virtualize and isolate OS resources (CPU, memory access, disk I/O, network access, user/group namespaces), so that processes inside the container see only a specific, virtualized set of resources
  - Linux control groups (cgroups) to control and limit the use of these resources
  - overlay filesystems to enable the appearance of writing to otherwise read-only filesystems
- Much more lightweight than a VM, as containers use many resources of the host's OS
- In a sense, give you control of your software environment without being root on the host system
  - with a catch: building containers from scratch usually requires root
- Ideal way to create a portable software environment: scientific software + all the dependencies


Docker image
in the cloud
Singularity image
in the cloud
apptainer pull image.sif <source>
existing SIF image.sif
write a def file
apptainer
shell
exec
run
image.sif
<source>
(nothing)
<command>
sudo apptainer build image.sif file.def
can use
--fakeroot
can use
--remote
apptainer build --sandbox image.dir <source>
sudo apptainer shell --writable ubuntu.dir
Apptainer> install packages, modify things
Apptainer> exit
sudo apptainer build image.sif image.dir
/bin/rm -rf image.dir
copy image to your destination
can use
--fakeroot
can use
--remote

Creating and mounting an overlay

- A writable overlay sits on top of an immutable SIF container image, can be:
  - a sandbox directory (does not work for our purposes)
  - ✔ a standalone writable ext3 filesystem image
  - a writable ext3 image embedded into the SIF file (less flexible)

```
cd ~/scratch/containers
module load apptainer/1.1.3
apptainer pull ubuntu.sif docker://ubuntu:latest
apptainer overlay create --size 512 small.img   # create a 0.5GB overlay image file

apptainer shell --overlay ./small.img ubuntu.sif
Apptainer> df -kh                  # the overlay should be mounted inside the container
Apptainer> mkdir -p /data          # a new root-level dir will go into the overlay image
Apptainer> cd /data && df -kh .    # using overlay; check for available space
Apptainer> for num in $(seq -w 00 19); do
               echo $num   # next line generates a binary file (1-33)MB in size
               dd if=/dev/urandom of=test"$num" bs=1024 count=$(( RANDOM + 1024 ))
           done
Apptainer> df -kh .          # should take ~350 MB
Apptainer> exit

apptainer shell --overlay ./small.img ubuntu.sif
Apptainer> ls /data          # here is your data
```

Creating and mounting an overlay (cont.)

- You can create an overlay image with a directory inside with something like this:

```
apptainer overlay create --create-dir /data --size 512 overlay.img
```

- You can mount the overlay in the read-only mode:

```
apptainer shell --overlay ./small.img:ro -B /home,/scratch ubuntu.sif
Apptainer>  touch /data/test.txt                # error: read-only file system
```

- To see the help page:

```
apptainer help overlay create
```

- Outside the container, when an overlay is *not* in use, you can resize it:

```
e2fsck -f small.img          # good idea to check your overlay's filesystem first
resize2fs -p small.img 1G    # resize your overlay
```

Sparse overlay images

- Empty blocks inside are not stored ⇒ use disk more efficiently
- Be careful using them: not all tools (e.g. backup/restore, `scp`, `sftp`, `gunzip`) recognize sparse files! ⇒ this can potentially lead to very bad things

1. Create a sparse overlay image:

```
apptainer overlay create --create-dir /data --size 512 --sparse sparse.img
ls -l sparse.img                    # its apparent size is 512MB
du -h --apparent-size sparse.img    # same
du -h sparse.img                    # its actual size is much smaller (17MB)
```

2. Mount it and fill with some data:

```
apptainer shell --overlay sparse.img ubuntu.sif

Apptainer> cd /data
Apptainer> for num in $(seq -w 00 19); do
               echo $num   # next line generates a binary file (1-33)MB in size
               dd if=/dev/urandom of=test"$num" bs=1024 count=$(( RANDOM + 1024 ))
           done
Apptainer> df -kh .         # should take just under 400MB, pay attention to "Used"

du -h sparse.img            # actual usage is now just under 400MB
```

Conda apptainer

- One possible use: install Conda into an overlay image
  - native Anaconda on our clusters is a bad idea https://docs.alliancecan.ca/wiki/Anaconda/en
  - installing into an overlay image takes a couple of minutes, results in over 17,000 files
  - no need for root, as you don't modify the container!
  - might still not be the most efficient use of resources (non-optimized binaries)

```
cd ~/scratch/containers
apptainer pull ubuntu.sif docker://ubuntu:latest
apptainer overlay create --size 550 --sparse conda.img   # create a 550MB overlay image
wget https://repo.anaconda.com/miniconda/Miniconda3-py39_4.12.0-Linux-x86_64.sh # to host

apptainer shell --overlay ./conda.img -B /scratch ubuntu.sif
Apptainer> mkdir /conda && cd /conda && df -kh .
Apptainer> bash ~/scratch/containers/Miniconda3-py39_4.12.0-Linux-x86_64.sh
            agree to the license
            use /conda/miniconda3 for the installation path
            "no" to initialize Miniconda3, to avoid modifying your host's ~/.bashrc
Apptainer> find /conda/miniconda3/ -type f | wc -l       # 17,722 files

du -h conda.img                                          # uses 474M
apptainer shell --overlay ./conda.img -B /scratch ubuntu.sif
Apptainer> source /conda/miniconda3/bin/activate
Apptainer> python                                        # works
```

Conda apptainer: installing Python packages into the overlay

```
# need more space if installing large packages
apptainer overlay create --size 2000 --sparse conda.img   # need almost 2G for numpy
apptainer shell --overlay ./conda.img -B /scratch ubuntu.sif
... install Conda as described in the previous slide ...

# -C flag important to force writing config files locally, and not to the host
apptainer shell -C --overlay ./conda.img -B /scratch ubuntu.sif
Apptainer> cd /conda/miniconda3
Apptainer> source bin/activate
Apptainer> conda install numpy
Apptainer> du -skh .    # used 1.5G
Apptainer> python
>>> import numpy as np
```

MPI codes in a container

There are fundamentally three different ways to run parallel MPI codes in a container:

1. Rely on MPI inside the container, e.g.

```
apptainer exec -B ... --pwd ... container.sif mpirun -np $SLURM_NTASKS ./mpicode
```

   - single Apptainer process running on the node
   - cgroup limitations from the Slurm job are passed into the container ⇒ set the number of available CPU cores ⇒ `mpirun` uses all available (in this job) CPU cores
   - ✘ limited to a single node
   - ✔ no need to adapt container's MPI to the host; just install SSH into the container

2. Hybrid mode: rely on `mpirun` on the host, and MPI inside the container to provide runtime MPI libraries, e.g.

```
mpirun -np $SLURM_NTASKS apptainer exec -B ... --pwd ... container.sif ./mpicode
```

   - separate Apptainer process per each MPI rank
   - ✔ can span multiple nodes
   - ✘ container's MPI should be customized to the host (not that difficult)

3. Bind mode - not used in this presentation

Option 1: build a container with precompiled OpenFOAM

- Easy:
  1. start with a stock `docker://ubuntu` image – already includes MPI
  2. install `openfoam10` package from `http://dl.openfoam.org/ubuntu`
- Works well with MPI inside the container, can scale OpenFOAM up to an entire node
- In general, will not play nicely with host's MPI and Slurm, so will not be able to do

```
mpirun -np $SLURM_NTASKS apptainer exec -B ... --pwd ... container.sif ./mpicode
```

- Instructions on building the container, preparing and running a small parallel OpenFOAM simulation while writing output to an overlay

  `https://github.com/razoumov/sharedSnippets/blob/main/openfoamPrecompiled.md`

Option 2: build a host-compatible OpenFOAM container (that can work in either mode)

1. Build an MPI container (that can talk to the host's MPI)

---

2. Run a test code with MPI entirely inside the container
3. Compile parallel OpenFOAM into this container
4. Run parallel OpenFOAM with container's MPI and with the case directory (and results) inside an overlay - limited to a single node

---

5. Run a test code with hybrid approach: host's MPI to launch the code, container's MPI for runtime libraries
6. Run parallel OpenFOAM in hybrid mode: organize output into multiple overlays
7. Post-process the results in the overlays

Step 1: build an MPI container

(that's compatible with host's Slurm and MPI)

- On the host, find out the supported MPI model:

```
srun --mpi=list                          # list the supported PMI implementations
scontrol show config | grep MpiDefault   # show the default
```

  e.g. on Arbutus Slurm supports pmi2 $\Rightarrow$ inside the container build OpenMPI with pmi2 support

- Create a definition file that:
  - bootstraps from `docker://ubuntu:22.04`
  - installs the necessary fabric and PMI packages, Slurm client, few others requirements
  - compiles the latest OpenMPI from source supporting hosts's PMI implementation
  - instructions for building a container to run on Cedar, Magic Castle clusters on Arbutus `https://github.com/razoumov/sharedSnippets/blob/main/mpiContainer.md`

- Build the container

```
sudo apptainer build mpi.sif parallelContainer.def
ls -lh mpi.sif    # 294M
scp mpi.sif username@cluster:path
```

Step 2: test a simple code inside the container

- Run a test code with container's `mpirun`

```
cd ~/scratch/containers/
module load openmpi apptainer/1.1.3
apptainer exec -B /scratch mpi.sif mpicc -O2 distributedPi.c -o distributedPi
salloc --nodes=1 --ntasks=3 --time=0:5:0 --mem-per-cpu=3600
apptainer exec -B /scratch mpi.sif mpirun -np $SLURM_NTASKS ./distributedPi
```

Step 3: compile parallel OpenFOAM into this container

OpenFOAM v2012 has a very subtle bug, resulting in an MPI deadlock in some of the runs with overlays $\Rightarrow$ using the previous major release v1912

```
sudo apptainer build --sandbox 1912.dir mpi.sif   # convert to a sandbox
sudo apptainer shell --writable 1912.dir

mkdir -p /opt/openfoam && cd /opt/openfoam
wget https://dl.openfoam.com/source/v1912/OpenFOAM-v1912.tgz
tar xvfz OpenFOAM-v1912.tgz && /bin/rm OpenFOAM-v1912.tgz
cd OpenFOAM-v1912
apt install -y flex libz-dev
WM_COMPILER_TYPE=ThirdParty
source etc/bashrc
./Allwmake -j 8        # took 1h35m
exit

sudo apptainer build openfoam1912.sif 1912.dir   # rebuild the container
scp openfoam1912.sif username@cluster:path
```

Step 4: prepare and run parallel OpenFOAM with container's MPI

and with the case directory (and results) inside an overlay - limited to a single node

```
module load openmpi apptainer/1.1.3   &&   cd ~/scratch/containers
apptainer overlay create --size 100 --create-dir /results output.img
apptainer shell -B /scratch --overlay output.img openfoam1912.sif

. /opt/openfoam/OpenFOAM-v1912/etc/bashrc
cd /results
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/motorBike .   # parallel case
cd motorBike
mv system/decomposeParDict.6 system/decomposeParDict
>>> edit system/decomposeParDict
      numberOfSubdomains  3;
      n (3 1 1);
>>> edit system/controlDict
      endTime         500;
      writeInterval   10;
      writeFormat     binary;
mv 0.orig 0
blockMesh      # generate the mesh
decomposePar   # decompose the mesh and the ICs
exit

export OPENFOAM=/opt/openfoam/OpenFOAM-v1912/platforms/linux64GccDPInt32Opt
export APPTAINERENV_PREPEND_PATH=$OPENFOAM/bin
export APPTAINERENV_LD_LIBRARY_PATH=$OPENFOAM/lib:$OPENFOAM/lib/openmpi-system
export APPTAINERENV_WM_PROJECT_DIR=/opt/openfoam/OpenFOAM-v1912

salloc --nodes=1 --ntasks=3 --time=0:10:0 --mem-per-cpu=3600
apptainer exec -B /scratch --overlay output.img --pwd /results/motorBike openfoam1912.sif \
  mpirun -np $SLURM_NTASKS simpleFoam -parallel
exit the job
```

Step 4 (cont.): analyze results

```
salloc --ntasks=1 --time=0:10:0 --mem-per-cpu=3600
apptainer shell -B /scratch --overlay output.img openfoam1912.sif

cd /results/motorBike
ls processor*
find processor? -type f | wc -l       # 1542 decomposed files including the setup and ICs
reconstructPar                        # merge decomposed files
find . -type f | wc -l                # 2476 files in total
rm -rf processor?/??? processor?/??   # delete the output decomposed files
find . -type f | wc -l                # still 976 files including 50 outputs with 10 files each
exit the container

exit the job
```

- A typical production run would have ∼1500 timesteps × 128 cores × ∼10 files/core/step
  ⇒ 1,920,000 decomposed files ⇒ great to have these inside an overlay
- How about scaling to multiple nodes?

Complication

- To scale to multiple nodes, you will use host's MPI, launching a separate container per each MPI rank

```
mpirun -np $SLURM_NTASKS apptainer exec -B /scratch --pwd ... \
  openfoam1912.sif simpleFoam -parallel
```

... but you cannot mount the same overlay file in multiple containers at the same time
... especially if these containers are spread across multiple nodes

Complication

- To scale to multiple nodes, you will use host's MPI, launching a separate container per each MPI rank

```
mpirun -np $SLURM_NTASKS apptainer exec -B /scratch --pwd ... \
  openfoam1912.sif simpleFoam -parallel
```

... but you cannot mount the same overlay file in multiple containers at the same time
... especially if these containers are spread across multiple nodes

- Solution: use a separate overlay image for each container (i.e., each MPI rank)
  (idea picked up from `https://pawseysc.github.io`)
  - single output (case) directory on the host's filesystem
  - inside it, use symbolic links to redirect output from each MPI rank to its own overlay

⇒ this will trick the simulation code into thinking that it's writing into a uniform filesystem

Step 5: run a test code with hybrid approach

(host's MPI to launch the code, container's MPI for runtime libraries)

```
mkdir -p ~/.openmpi
echo "btl_vader_single_copy_mechanism=none" >> ~/.openmpi/mca-params.conf
module load openmpi apptainer/1.1.3
export PMIX_MCA_psec=native   # allow mpirun to use host's PMI
cd ~/scratch/containers
apptainer exec -B /scratch mpi.sif mpicc -O2 distributedPi.c -o distributedPi
salloc --ntasks=2 --time=0:5:0 --mem-per-cpu=3600
mpirun -np $SLURM_NTASKS apptainer exec -B /scratch mpi.sif ./distributedPi
```

Step 5: run a test code with hybrid approach

(host's MPI to launch the code, container's MPI for runtime libraries)

```
mkdir -p ~/.openmpi
echo "btl_vader_single_copy_mechanism=none" >> ~/.openmpi/mca-params.conf
module load openmpi apptainer/1.1.3
export PMIX_MCA_psec=native   # allow mpirun to use host's PMI
cd ~/scratch/containers
apptainer exec -B /scratch mpi.sif mpicc -O2 distributedPi.c -o distributedPi
salloc --ntasks=2 --time=0:5:0 --mem-per-cpu=3600
mpirun -np $SLURM_NTASKS apptainer exec -B /scratch mpi.sif ./distributedPi
```

Later we'll using `$SLURM_PROCID` variable from individual ranks

$\Rightarrow$ for that we'll use `srun`

```
# this breaks b/c of an environment bug
srun -n $SLURM_NTASKS apptainer exec -B /scratch mpi.sif ./distributedPi

# this works
echo "apptainer exec -B /scratch mpi.sif ./distributedPi" | srun -n $SLURM_NTASKS bash
# this prints each rank's ID
echo "apptainer exec mpi.sif echo "'$SLURM_PROCID' | srun -n $SLURM_NTASKS bash
```

Step 6a: run OpenFOAM in hybrid mode - prepare the simulation

```
cd ~/scratch/containers
module load openmpi apptainer/1.1.3
export PMIX_MCA_psec=native
/bin/rm -rf ~/scratch/OpenFOAM/*
apptainer shell -B /scratch --pwd ~/scratch/OpenFOAM openfoam1912.sif

. /opt/openfoam/OpenFOAM-v1912/etc/bashrc
mkdir -p ~/scratch/OpenFOAM/run && cd ~/scratch/OpenFOAM/run    # host's filesystem
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/motorBike .     # parallel case
cd motorBike
mv system/decomposeParDict.6 system/decomposeParDict
>>> edit system/decomposeParDict
  numberOfSubdomains  3;
  n (3 1 1);
>>> edit system/controlDict
  endTime         500;
  writeInterval   10;
  writeFormat     binary;
  runTimeModifiable false;   # disable loading of dictionaries at each timestep
mv 0.orig 0
blockMesh      # generate the mesh
decomposePar   # decompose the mesh (and the ICs)
exit
```

Step 6b: replace output dirs with links to equivalents in MPI overlays

```
cd ~/scratch/OpenFOAM/motorBike      # simulation directory on the host's filesystem
mkdir -p bak && mv processor* bak    # move them out of the way for now

cd ~/scratch/containers
# create a 100MB overlay
apptainer overlay create --size 100 --create-dir /results/motorBike results0.img
for rank in {1..2}; do
  cp results0.img results${rank}.img
done

for rank in {0..2}; do   # for each processor copy the initial conditions and the mesh
  apptainer exec -B /scratch --overlay results${rank}.img openfoam1912.sif \
    cp -r ~/scratch/OpenFOAM/motorBike/bak/processor${rank} /results/motorBike
done

cd ~/scratch/OpenFOAM/motorBike
for rank in {0..2}; do
  # these links will be broken on the host
  # but they will work inside each rank's container
  ln -s /results/motorBike/processor${rank} processor${rank}
done
```

How to see content inside output directories?

```
cd ~/scratch/containers
apptainer shell --overlay results0.img -B /scratch \
  --pwd ~/scratch/OpenFOAM/motorBike openfoam1912.sif
ls processor*   # only processor0 will show files
exit
>>> the same when mounting results1.img - only processor1 will show files, and so on ...

cd ~/scratch/containers
salloc --ntasks=3 --time=0:60:0 --mem-per-cpu=3600
# look inside each rank's directory
echo 'apptainer exec --overlay results${SLURM_PROCID}.img -B /scratch \
  --pwd ~/scratch/OpenFOAM/motorBike openfoam1912.sif \
  ls processor${SLURM_PROCID}/' | srun -n $SLURM_NTASKS bash
```

Step 6c: run the simulation

```
export OPENFOAM=/opt/openfoam/OpenFOAM-v1912/platforms/linux64GccDPInt32Opt
export APPTAINERENV_PREPEND_PATH=$OPENFOAM/bin
export APPTAINERENV_LD_LIBRARY_PATH=$OPENFOAM/lib:$OPENFOAM/lib/openmpi-system
export APPTAINERENV_WM_PROJECT_DIR=/opt/openfoam/OpenFOAM-v1912

salloc --ntasks=3 --time=0:30:0 --mem-per-cpu=3600

echo 'apptainer exec --overlay results${SLURM_PROCID}.img -B /scratch \
  --pwd ~/scratch/OpenFOAM/motorBike openfoam1912.sif simpleFoam \
  -parallel' | srun -n $SLURM_NTASKS bash

echo 'apptainer exec --overlay results${SLURM_PROCID}.img -B /scratch \
  --pwd ~/scratch/OpenFOAM/motorBike openfoam1912.sif \
  ls processor${SLURM_PROCID}/' | srun -n $SLURM_NTASKS bash     # check the results
```

Step 7: postprocessing – reconstruct ...

We can't mount more than one writable overlay per container ... but we can mount (1) multiple overlays as read-only + (2) one writable overlay:

```
cd ~/scratch/containers
apptainer overlay create --size 300 --create-dir /analyze reconstructed.img
salloc --time=0:30:0 --mem-per-cpu=3600
apptainer shell \
  --overlay $(echo results{0..2}.img:ro | tr ' ' ,) \   # per-rank results
  --overlay reconstructed.img \                         # reconstructed results
  -B /scratch --pwd ~/scratch/OpenFOAM/motorBike openfoam1912.sif

ls processor*    # see output from all ranks
mv  * /analyze/  # fast: not moving the results, just the initial setup + links
cd /analyze

reconstructPar
exit
apptainer shell --overlay reconstructed.img -B /scratch --pwd analyze openfoam1912.sif

ls               # here are your reconstructed results
ls processor*/   # these links are now are broken
```

Step 7: postprocessing – ... or not

1. Install/compile parallel ParaView server into another MPI container, call it `paraview.sif`
2. Run parallel ParaView server

```
cd ~/scratch/containers
salloc --ntasks=3 --time=0:60:0 --mem-per-cpu=3600
echo 'apptainer exec --overlay results${SLURM_PROCID}.img -B /scratch \
  --pwd ~/scratch/OpenFOAM/motorBike paraview.sif \
  pvserver --force-offscreen-rendering' | srun -n $SLURM_NTASKS bash
```

3. Take note of the listening node's name and port number

```
ssh <username>@<cluster>.computecanada.ca -L 11111:<node>:<port>
```

4. Connect from the ParaView client on your computer to `localhost:11111`
5. Create an empty `case.foam` file in the case directory
6. Load `case.foam`, set Case Type = Decomposed

Summary

- Overlays give us a great way to add writable layers to a container, no need for root to compile software into an overlay
- Can mix and match containers and overlays

Questions?

- Only one limitation: you can mount only one writable overlay per container at a time
  - ⇒ can mount many as read-only
  - ⇒ can mount one writable overlay per MPI rank (if running one container per MPI rank)
- The examples in these slides shown with `salloc`, but of course all production runs should be scheduled with `sbatch` and proper Slurm scripts
- In a parallel run, what if all output is happening on a single MPI rank?
  - ⇒ mount the overlay on this rank, need only one overlay
- In a parallel run, what if all your processors write their output into the same directory?
  - well, could they be doing parallel I/O, writing into a few large files?
  - if not, and you have the many-files problem ⇒ probably there is a custom solution for each code, talk to us
  - or modify your code's output, so that it does not create the problem in the first place, or make it write output files into per-processor directories + apply the solution from this presentation