Remote visualization on Compute Canada clusters

ALEX RAZOUMOV
alex.razoumov@westgrid.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@westgrid.ca
- Link to these slides below

Why remote visualization?

- ✘ Dataset could be too large to download
- ✘ Dataset and its analysis workflow cannot fit into your computer's memory
- ✘ Rendering is too slow on your computer: limited CPU/GPU power
- ✔ Added benefit: reading *compatible* data with parallel I/O is much faster

- Special use cases:
  - in-situ visualization = instrumenting a simulation code on the cluster to (1) output graphics or (2) connect to a visualization frontend (ParaView, VisIt) on the fly
  - web-based visualization with data served from another location
  - required visualization software is licensed or available only on Compute Canada systems (fortunately, not a common scenario!)

Remote visualization choices

Your data is on an HPC cluster.

You will be doing remote rendering.

$\Rightarrow$ You have several *orthogonal* rendering choices:

1. Interactive remote desktop vs. interactive client-server vs. batch (offscreen)
2. GPU vs. CPU + choice of several raytracers and/or rasterizers for the hardware
3. Serial vs. parallel
4. Using a local proxy for debugging vs. remote debugging

Batch rendering

Batch = offscreen (no GUI interaction)

1. commands in an interactive shell
2. precompiled rendering script, often submitted as a Slurm job

Usual benefits of scripting:

- automate mundane or repetitive or long tasks, e.g. making mutiple frames for a movie
- document your workflow, and make it reproducible
- do visualization on clusters without any GUI elements and/or much interactive resources, e.g., via a job scheduled from the command line

Workflow in any Linux-compatible visualization tool with a programming interface (in a compiled or interpreted language) can be scripted on a cluster

Open-source sci-vis tools with scripting interfaces

- Python libraries
  - basic 1D/2D/3D plotting with Matplotlib, Plotly, Bokeh
  - YT Python library for multi-resolution 3D volumetric rendering and analysis
- Many domain-specific packages support scripting, e.g.
  - VMD (Visual Molecular Dynamics) provides Python and Tcl interfaces
  - ... for batch visualization help with any domain-specific tool, please contact us
- General-purpose tools
  - VTK (Visualization Toolkit) library has C++, Tcl/Tk, Java, JavaScript and Python interfaces – can be used as a standalone renderer
  - Mayavi2, a serial 3D interactive scientific data visualization package, has an embedded Python shell
  - both ParaView and VisIT provide Python scripting (and compiled language interfaces for in-situ visualization)

Open-source sci-vis tools with scripting interfaces

- Python libraries
  - basic 1D/2D/3D plotting with **Matplotlib**, **Plotly**, Bokeh
  - YT Python library for multi-resolution 3D volumetric rendering and analysis
- Many domain-specific packages support scripting, e.g.
  - VMD (Visual Molecular Dynamics) provides Python and Tcl interfaces
  - ... for batch visualization help with any domain-specific tool, please contact us
- General-purpose tools
  - VTK (Visualization Toolkit) library has C++, Tcl/Tk, Java, JavaScript and Python interfaces – can be used as a standalone renderer
  - Mayavi2, a serial 3D interactive scientific data visualization package, has an embedded Python shell
  - both **ParaView** and **VisIT** provide Python scripting (and compiled language interfaces for in-situ visualization)

Installing Python packages into your virtual environment

- Initial setup:

```
$ module avail python                  # several versions available
$ module load python/3.8.10 mpi4py
$ virtualenv --no-download ~/astro     # install Python tools into your ~/astro
$ source ~/astro/bin/activate
(astro) $ pip install --no-index --upgrade pip
(astro) $ pip install --no-index matplotlib numpy plotly h5py
(astro) $ pip install yt
...
```

- Usual workflow:

```
$ module load mpi4py               # if you need it (for yt)
$ source ~/astro/bin/activate      # load the environment
(astro) $ python
...
(astro) $ deactivate
```

Matplotlib with a batch script

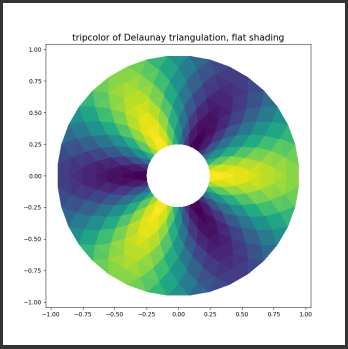
- In Python's matplotlib can script the entire workflow without opening windows
  1. use a "hardcopy" backend (offscreen output) such as PNG, SVG, PDF, PS
  2. save the image at the end
- More details at

  `https://matplotlib.org/stable/tutorials/introductory/usage.html#what-is-a-backend`
- See `https://matplotlib.org/stable/gallery/index.html` for plotting examples covering many 1D, 2D and simple 3D use cases
- Run on the command line (tiny visualizations on the login node) or submit to the scheduler via the script:

```
#!/bin/bash
#SBATCH --time=00:05:00   # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --mem=1000        # in MB
#SBATCH --account=def-someuser
source ~/astro/bin/activate
python triangulation.py
```

Matplotlib with a batch script (cont.)

Example adapted from `https://matplotlib.org/3.1.1/gallery`



Script `triangulation.py`

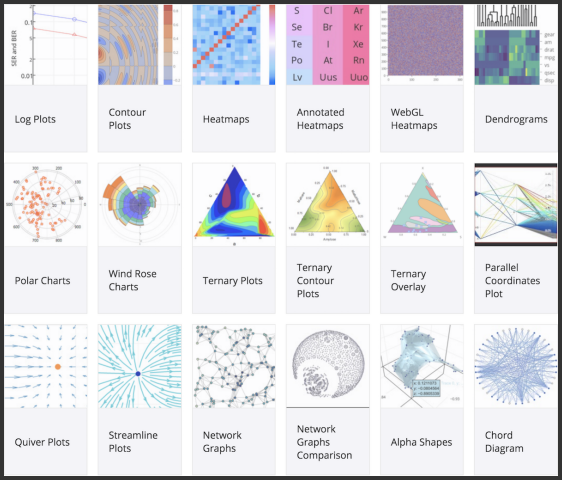
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
mpl.use('Agg')      # enable PNG backend

n_angles, n_radii, min_radius = 36, 8, 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2 * np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi / n_angles
x, y = (radii * np.cos(angles)).flatten(), (radii * np.sin(angles)).flatten()
z = (np.cos(radii) * np.cos(3 * angles)).flatten()
triang = tri.Triangulation(x, y) # create Delaunay triangulation
triang.set_mask(np.hypot(x[triang.triangles].mean(axis=1),    # mask off unwanted triangles
                         y[triang.triangles].mean(axis=1)) < min_radius)

plt.figure(figsize=(8,8))
plt.tripcolor(triang, z, shading='flat')    # pseudocolor of unstructured triangular grid
plt.title('tripcolor of Delaunay triangulation, flat shading', fontdict = {'fontsize' : 15})
plt.savefig('delaunay.png')
```

Plotly Python library

- Open-source project from Plot.ly
  `https://plot.ly/python`
- Produces dynamic html5 visualizations for the web
- APIs for Python (with/without Jupyter), R, JavaScript, MATLAB

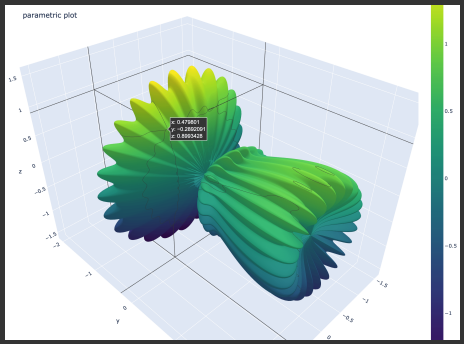


- Can work offline (free) or by sending your data to your account on plot.ly (limited public plotting is free, paid unlimited private plotting and extra tools)

Plotly with a batch script

Example adapted from Plotly tutorials

1. Offline plotting into a file
2. Set `auto_open=False`



Script `parametric.py`

```
import plotly.offline as py    # offline plotting
import plotly.graph_objs as go
from numpy import pi, sin, cos, mgrid
dphi, dtheta = pi/250, pi/250    # 0.72 degrees

# define two 2D grids: both phi and theta are 252x502 numpy arrays
[phi, theta] = mgrid[0:pi+dphi*1.5:dphi, 0:2*pi+dtheta*1.5:dtheta]
r = sin(4*phi)**3 + cos(2*phi)**3 + sin(6*theta)**2 + cos(6*theta)**4
x = r*sin(phi)*cos(theta)    # x is also 252x502
y = r*cos(phi)               # y is also 252x502
z = r*sin(phi)*sin(theta)    # z is also 252x502

surface = go.Surface(x=x, y=y, z=z, colorscale='Viridis')
layout = go.Layout(title='parametric plot')
fig = go.Figure(data=[surface], layout=layout)
py.plot(fig, filename='parametric.html', auto_open=False)
```

Visualizing 3D volumetric data with HTTP://YT-PROJECT.ORG

- Python package for analyzing and visualizing volumetric, multi-resolution data
  - library for non-interactive use, no 3D interactivity found in such tools as ParaView and VisIt
    - there is an ongoing project VIEWYT to develop Qt widgets for interacting with YT plots
  - discretization: structured, unstructured, variable-resolution (curvilinear), particle data
  - slice and projection plots
  - volume rendering with full control of scene setup, camera position, transfer function
  - very easy to learn, wonderful documentation at `https://yt-project.org/doc`
  - great for batch off-screen rendering (including HPC clusters); parallelized with `mpi4py`
- Initially written for analysing *Enzo* output data, adapted to understand other data formats from astrophysics and beyond
  - documentation strongly focused on astrophysical data (do not let this deter you)
  - currently has readers for $\sim$ 25 file formats
  - can import generic data on uniform and AMR/nested grids, particles, unstructured meshes
  - popular in astrophysics, seismology, nuclear engineering, molecular dynamics, oceanography
- Watch our 2-part webinar series "Using YT for analysis and visualization of volumetric data" (part 1) and "Working with data objects in YT" (part 2) at `http://bit.ly/vispages`

Rotating cosmological volume with grid annotations in YT

More on parallel YT at https://yt-project.org/doc/analyzing/parallel_computation.html

1. Download/uncompress the data from http://yt-project.org/data
2. On the cluster, save this as grids.py:

```
import yt, numpy as np
yt.enable_parallelism()   # turn on MPI parallelism via mpi4py
ds = yt.load("Enzo_64/DD0043/data0043")
sc = yt.create_scene(ds, ('gas', 'density'))
cam = sc.camera
cam.resolution = (1024, 1024)           # resolution of each frame
sc.annotate_domain(ds, color=[1, 1, 1, 0.005])   # draw the domain boundary [r,g,b,alpha]
sc.annotate_grids(ds, alpha=0.005)      # draw grid boundaries
sc.save('frame0000.png', sigma_clip=4)
for i in cam.iter_rotate(np.pi, 900):   # rotate by 180 degrees over 900 frames
    sc.save('frame%04d.png' % (i+1), sigma_clip=4)
```

3. Write the job submission script yt-mpi.sh:

```
#!/bin/bash
#SBATCH --time=12:00:00   # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --ntasks=4        # number of MPI processes
#SBATCH --mem-per-cpu=3600
#SBATCH --account=def-someuser
module load mpi4py        # important to load this before next line
source ~/astro/bin/activate
srun python grids.py
```

Rotating cosmological volume with grid annotations in YT (cont.)

- Submit the job

```
$ sbatch yt-mpi.sh
```

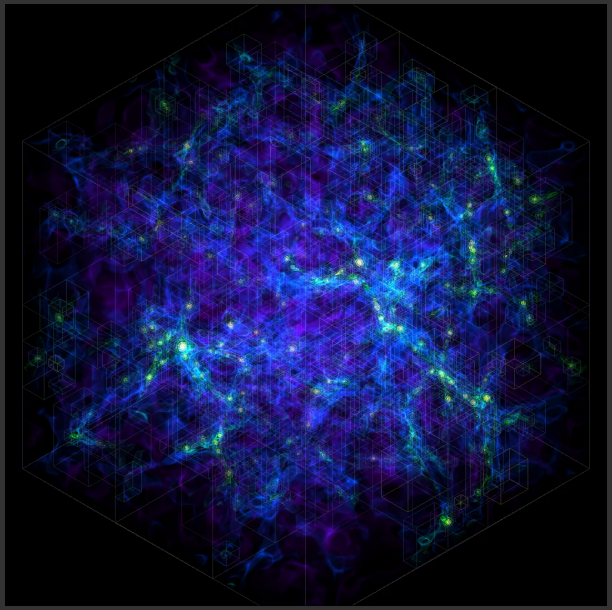
- Performance: serial ➢ 1.47 frames/min., parallel on 4 cores ➢ 4.05 frames/min.
- Make a Quicktime-compatible MP4 right on the cluster

```
$ ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p -vf "
    scale=trunc(iw/2)*2:trunc(ih/2)*2" grids.mp4
```

- Download it to your laptop

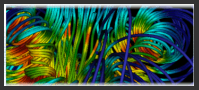
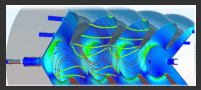
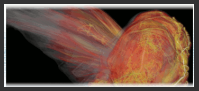
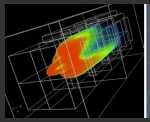
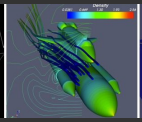
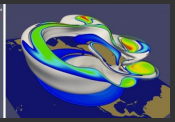
```
$ rsync -av --progress cedar.computecanada.ca:path/to/grids.mp4 .
```

Rotating cosmological volume with grid annotations in YT (cont.)



- Online (highly compressed)
  `https://vimeo.com/301503962`
- On presenter's laptop `grids.mp4`

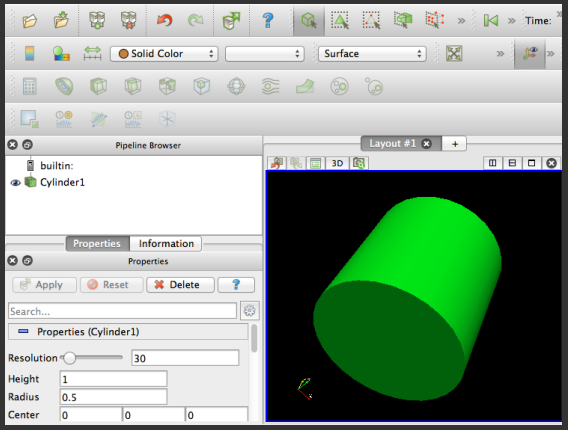
ParaView and VisIT

- Built to visualize extremely large (GBs to TBs) datasets on distributed-memory machines
- Scale to many ($10^3 - 10^5$) cores via MPI
- Work equally well on a laptop
- Open source, under active development
- Multi-platform (Linux / Mac / Windows)
- Scalar, vector, tensor fields
- Wide variety of discretizations: structured, unstructured, particles, irregular in 2D/3D
- Nice interactive GUI and Python scripting
- Client-server and batch offscreen modes
- Support over 100 input data formats
- Support parallel I/O
- Huge array of visualization features
- Based on VTK

I assume you are already familiar with ParaView basics

- main elements of the GUI
- how to load a dataset (and convert your data into a Paraview-readable format)
- how to switch between different views (representations)
- how to colour visualization by a variable
- how to use filters to build a pipeline



If not, see our regular ParaView workshop materials at `https://bit.ly/paraviewzipp` (slides, datasets and codes in one ZIP file) and `http://www.paraview.org/documentation`

Remote visualization

If your dataset is on cluster.computecanada.ca, you have several options:

✘ download data to your desktop and visualize it locally

✘ run ParaView remotely on the cluster via X11 forwarding
your desktop $\xrightarrow{\text{ssh } -X}$ larger machine running ParaView

✔ run ParaView remotely on the cluster **via remote desktop**
your desktop $\xrightarrow{\text{VNC}}$ larger machine running ParaView
$\Rightarrow$ see next slide

✔ run ParaView in **client-server mode**
ParaView client on your desktop $\rightleftharpoons$ ParaView server on the cluster
$\Rightarrow$ see the following slide

✔ run ParaView via a **GUI-less batch script** (interactively or scheduled)

For remote options (2) - (5), some setup details may vary across the systems

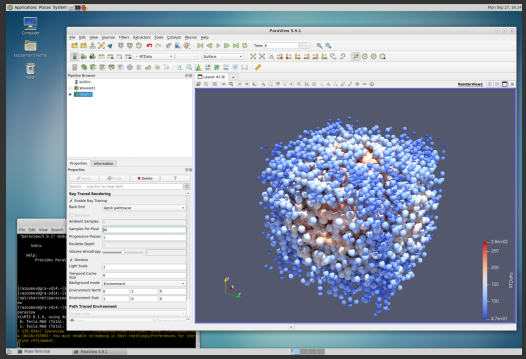
- render server can run with or without GPU rendering
- data/render servers can run on single-core, or across several cores/nodes with MPI
- for interactive GUI work on clusters it's best to schedule interactive jobs, as opposed to running on the login nodes

ParaView via remote desktop

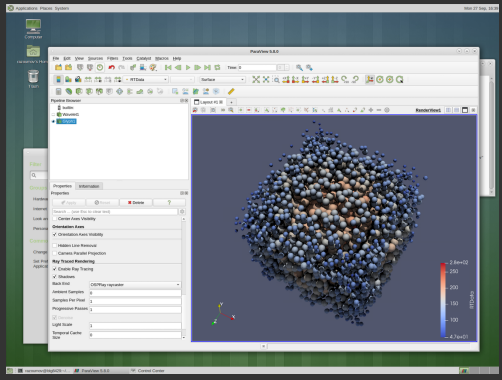
`https://docs.computecanada.ca/wiki/VNC`

(1) run a VNC server on compute nodes with SSH tunnelling, connect via a VNC client

`https://docs.computecanada.ca/wiki/VNC#Compute_Nodes`



(2) VNC on gra-vdi.computecanada.ca, connect via a VNC client



(3) Remote Desktop via JupyterHub on Béluga, point your web browser at `https://jupyterhub.beluga.computecanada.ca`

ParaView's distributed parallel architecture

Three logical units of ParaView – these units can be embedded in the same application on the same computer, but can also run on different machines:

- **Data Server** – The unit responsible for data reading, filtering, and writing. All of the pipeline objects seen in the pipeline browser are contained in the data server. The data server can be parallel.
- **Render Server** – The unit responsible for rendering. The render server can also be parallel, in which case built-in parallel rendering is also enabled.
- **Client** – The unit responsible for establishing visualization. The client controls the object creation, execution, and destruction in the servers, but does not contain any of the data, allowing the servers to scale without bottlenecking on the client. If there is a GUI, that is also in the client. The client is always a serial application.

Remote ParaView workflow 1: small-scale visualization with local debugging and remote batch rendering

1. Debug serial visualization script in standalone ParaView on a laptop
2. Run it locally from the GUI and from the command line
3. Modify it to create animation
4. Copy the script to a cluster and run it there on a login node
5. Run the script with a serial Slurm job

Example 1

3D sine envelope wave function defined inside a unit cube ($x_i \in [0, 1]$)

$$f(x_1, x_2, x_3) = \sum_{i=1}^{2} \left[\frac{\sin^2\left(\sqrt{\xi_{i+1}^2 + \xi_i^2}\right) - 0.5}{\left[0.001(\xi_{i+1}^2 + \xi_i^2) + 1\right]^2} + 0.5 \right], \text{ where } \xi_i \equiv 30(x_i - 0.5)$$

discretized on a 100^3 Cartesian grid

- You'll find the code `generateSineEnvelope.py` inside the ZIP file

```
$ conda install numpy netcdf4     # on your computer
$ python generateSineEnvelope.py
```

- This will produce the file `sineEnvelope.nc` that ParaView can read
  - you can also find it in our regular ParaView workshop materials `http://bit.ly/paraviewzipp` under `data/`

Demo time

1. Run `generateSineEnvelope.py` ⇒ this will produce `sineEnvelope.nc`
2. Start ParaView 5.8 on my laptop (to match `paraview-offscreen/5.8.0`)
3. Tools → Start Trace
4. Load `sineEnvelope.nc`
5. Switch to Volume representation, edit transfer function and colourmap
6. File → Save Screenshot... as PNG file
7. Tools → Stop Trace and save the script as `volume.py`

Running the script

- For your convenience, I added the optimized version `volume1.py` to the ZIP file
- Multiple ways to run this script:
  - open ParaView's built-in Python interpreter View → Python Shell, then Run Script
  - *[/usr/bin/ /usr/local/bin/ /Applications/ParaView-5.8.1.app/Contents/bin/]*
    pvpython will give you a Python shell connected to a ParaView server without the GUI – can copy and paste commands here
  - *[/usr/bin/ /usr/local/bin/ /Applications/ParaView-5.8.1.app/Contents/bin/]*
    pvbatch *volume1.py* will run the script

Extending the script

Typing commands inside ParaView's Python shell after running the script

```
>>> help(GetActiveCamera)
Help on function GetActiveCamera in module paraview.simple:

GetActiveCamera()
    Returns the active camera for the active view. The returned object
    is an instance of vtkCamera.

>>> dir(GetActiveCamera())   # list all the fields and methods of the object
['AddObserver', 'ApplyTransform', 'Azimuth', 'BreakOnError', 'ComputeViewPlaneNormal', '
    DebugOff', 'DebugOn', 'DeepCopy', 'Dolly', 'Elevation', 'FastDelete', '
    GetAddressAsString', 'GetCameraLightTransformMatrix', 'GetClassName', '
    GetClippingRange', 'GetCommand', 'GetCompositeProjectionTransformMatrix', 'GetDebug',
    'GetDirectionOfProjection', 'GetDistance', 'GetEyeAngle', 'GetEyePlaneNormal', '
    GetEyePosition', 'GetEyeSeparation', 'GetEyeTransformMatrix', 'GetFocalDisk', '
    GetFocalPoint', 'GetFreezeFocalPoint', 'GetFrustumPlanes', 'GetGlobalWarningDisplay',
    'GetLeftEye', 'GetMTime', 'GetModelTransformMatrix', 'GetModelViewTransformMatrix', '
    GetModelViewTransformObject', 'GetOrientation', 'GetOrientationWXYZ', '
    GetParallelProjection', 'GetParallelScale', 'GetPosition', '
    GetProjectionTransformMatrix', 'GetProjectionTransformObject', 'GetReferenceCount', '
    GetRoll', 'GetScreenBottomLeft', 'GetScreenBottomRight', 'GetScreenTopRight', '
    GetThickness', 'GetUseHorizontalViewAngle', 'GetUseOffAxisProjection', '
    GetUserTransform', 'GetUserViewTransform', 'GetViewAngle', ...]
```

Extending the script (cont.)

```
>>> help(GetActiveCamera().Azimuth)
Help on built-in function Azimuth:

Azimuth(...)
    V.Azimuth(float)
    C++: void Azimuth(double angle)

    Rotate the camera about the view up vector centered at the focal
    point. Note that the view up vector is whatever was set via
    SetViewUp, and is not necessarily perpendicular to the direction
    of projection.  The result is a horizontal rotation of the
    camera.
```

Extending the script (cont.)

Let's replace the current line

```
SaveScreenshot('volume.png', renderView1, ImageResolution=[1100, 768])
```

inside `volume1.py` with the following:

```
camera = GetActiveCamera()
numberFrames = 90
for i in range(numberFrames):
    camera.Azimuth(1)     # rotate by 1 degree
    print('writing frame%04d'%(i)+'.png')
    SaveScreenshot(path+'frame%04d'%(i)+'.png', renderView1, ImageResolution=[1100,
        768])
```

save it as `volume2.py`, and run the script again

```
$ /path/to/pvbatch volume2.py
```

Creating local animation

- This will produce 90 files `frame{0000..0089}.png` each rotated by a one degree compared to the previous one
- Can merge them into a movie with a third-party tool, e.g.

```
$ ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p -vf "scale=
    trunc(iw/2)*2:trunc(ih/2)*2" volume.mp4
$ ls -lh volume.mp4
-rw-r--r--@ 1 razoumov  staff   273K 14 Sep 14:35 volume.mp4
```

Remote script execution on the login node

```
[laptop]$ scp sineEnvelope.nc volume2.py razoumov@cedar.computecanada.ca:scratch/webinar
    /

[cedar1]$ cd ~/scratch/webinar
[cedar1]$ sed -i -e 's|/Users/razoumov/Documents/09-remoteVisWebinar|/scratch/razoumov/
    webinar|' volume2.py
[cedar1]$ module load gcc/9.3.0 paraview-offscreen/5.8.0
[cedar1]$ pvbatch --force-offscreen-rendering volume2.py
[cedar1]$ ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p -vf "scale=trunc(
    iw/2)*2:trunc(ih/2)*2" remoteVolume.mp4

[laptop]$ scp razoumov@cedar.computecanada.ca:scratch/webinar/remoteVolume.mp4 .
```

- Not opening any remote graphics windows, just shipping the final movie (tiny file!)
  - not using X11 forwarding (unnecessary network traffic)
  - no need for an X11 server on your laptop
- Please don't do any heavy graphics on the login node(s) as it is shared among many users!

Remote script execution on compute nodes

```
from paraview.simple import *
sineEnvelopenc = NetCDFReader(FileName=['/scratch/razoumov/webinar/sineEnvelope.nc'])
renderView1 = GetActiveViewOrCreate('RenderView')
renderView1.CameraPosition, renderView1.CameraFocalPoint = [-70.9, 194.6, 321.9], [49.5, 49.5, 49.5]
renderView1.CameraViewUp, renderView1.CameraParallelScale = [0.08354, 0.89439, -0.43940], 90.
sineEnvelopencDisplay = Show(sineEnvelopenc, renderView1)
sineEnvelopencDisplay.SetRepresentationType('Volume')
densityLUT, densityPWF = GetColorTransferFunction('density'), GetOpacityTransferFunction('density')
densityPWF.Points = [0.01944, 0.0, 0.5, 0.0, 1.521604, 0.0, 0.5, 0.0, 1.99497, 1.0, 0.5, 0.0]
densityLUT.ApplyPreset('Cold and Hot', True)            # colourmap
camera = GetActiveCamera()
for i in range(90):
    camera.Azimuth(1)    # rotate by 1 degree
    print('writing frame%04d'%(i)+'.png')
    SaveScreenshot('/scratch/razoumov/tmp/frame%04d'%(i)+'.png', renderView1, ImageResolution=[1100, 768])
```

```
#!/bin/bash
#SBATCH --time=00:05:00   # walltime in d-hh:mm or hh:mm:ss format
#SBATCH --mem=3600        # in MB
#SBATCH --account=def-someuser
module load gcc/9.3.0 paraview-offscreen/5.8.0
pvbatch --force-offscreen-rendering volume2.py
ffmpeg -r 30 -i frame%04d.png -c:v libx264 -pix_fmt yuv420p -vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" \
       remoteVolume.mp4
/bin/rm -f frame????.png
```

```
$ cd ~/scratch/webinar
$ sbatch paraview-serial.sh
```

Remote script execution on compute nodes (cont.)

How does this work on a GPU-less node?

Answer: `paraview-offscreen/5.8.0` was compiled with OSMesa off-screen rendering library (software-based OpenGL without X11 dependency)

- Different from OSPRay (Intel's CPU-based ray tracing library)
- We will talk about GPU rendering later

Remote ParaView workflow 2:

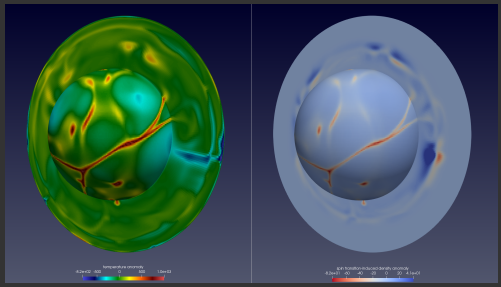
small-scale visualization with remote client-server debugging

1. Debug serial visualization script in client-server mode (remote debugging)
   - cannot run the script locally (no data) $\Rightarrow$ can only debug on the cluster
2. Simplify the script on your laptop with a text editor, but do not run it locally
3. Copy the script to a cluster and run it there inside a serial interactive job
4. Run the script with a serial Slurm job

Example 2: Earth's mantle convection

Dataset from IEEE 2021 SciVis Contest `https://scivis2021.netlify.app`

- Dataset from *Earth's Mantle Convection* simulation by Hosein Shahnas and Russell Pysklywec (U. of Toronto)
  - dataset details at `https://scivis2021.netlify.app/data`
  - 251 timesteps on a spherical $180 \times 201 \times 360$ grid
- You can render this dataset in serial



- Data in `cedar:/project/6003910/razoumov/ieeevis2021/spherical` (89GB in total; each timestep just under 400MB)

Demo time

1. On the cluster start remote serial ParaView server:

```
$ cd scratch/webinar   # necessary on Cedar
$ salloc --time=0:60:0 --mem-per-cpu=8000 --account=def-someuser --partition=
    cpubase_interac
$ module load gcc/9.3.0 paraview-offscreen/5.8.0
$ pvserver
```

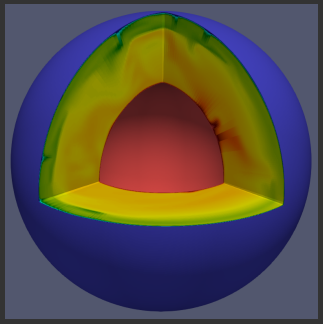
2. Wait for it to start waiting for incoming connection:

```
Waiting for client...
Connection URL: cs://cdr767.int.cedar.computecanada.ca:11111
Accepting connection(s): cdr767.int.cedar.computecanada.ca:11111
```

3. On your laptop start SSH port forwarding:

```
$ ssh cedar.computecanada.ca -L 11111:cdr767:11111   # use the actual compute node
```

4. On your laptop start ParaView 5.8.x, click Connect, then connect to `cs://localhost:11111`



5. Tools → Start Trace
6. Load one of `spherical???.nc`
7. Clip with a Box, uncheck Invert if needed
8. Colour by temperature
9. Save the image as a PNG file
   - in client-server it'll be saved as a local file on your computer
   - in production workflow you will be saving to the cluster's filesystem
10. Tools → Stop Trace
11. Save the generated script as `mantle.py` locally
    - edit it in a text editor, simplify (most lines will be setting defaults), but do not run it locally
    - provide the correct output PNG path on the remote system
    - I put the optimized version `mantle2.py` inside the ZIP file

12. Upload the script to the cluster:

```
$ scp mantle2.py cedar.computecanada.ca:scratch/webinar/
```

13. On the cluster try running it as a parallel interactive job:

```
$ cd ~/scratch/webinar
$ salloc --time=0:60:0 --mem-per-cpu=8000 --account=def-someuser --partition=
    cpubase_interac
$ module load gcc/9.3.0 paraview-offscreen/5.8.0
$ pvbatch --force-offscreen-rendering mantle2.py
```

14. Once you are happy with the result, write a Slurm job submission script and submit it with `sbatch`

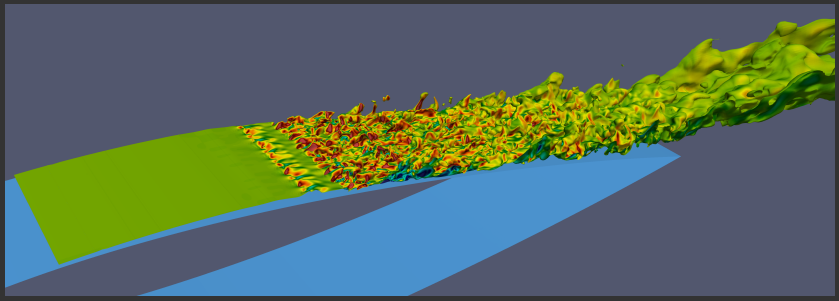
Remote ParaView workflow 3: large-scale visualization with remote debugging

1. Debug parallel visualization script in client-server mode (remote debugging)
   - cannot run the script locally (data too large) $\Rightarrow$ can only debug on the cluster
2. Simplify the script on your laptop with a text editor
3. Copy the script to the cluster and run it there inside a parallel interactive job
4. Modify it to create animation
5. Eventually do parallel batch rendering with Slurm

Distributed-memory (parallel) rendering

https://computecanada.github.io/visualizeThis

Let's render the "default" CFD dataset from 2019 *Visualize This!*



- OpenFOAM *decomposed* dataset: 512 cores, 5 hydro variables, ~959GB = mesh + 86 timesteps
  - kindly provided for this competition by Joshua Brinkerhoff (UBC Okanagan)
  - unstructured mesh ⇒ loading a single timestep from the 3D internal mesh requires 200GB+ physical RAM
  - the 2D airfoil mesh takes only 13.7 GB virtual memory for 1 timestep + 1 variable
- Image above shows the isosurface of constant air speed coloured by the Y-component of the vorticity, full animation rendering (86 timesteps) took 17 minutes on 128 Cedar CPU cores

Demo time

1. On the cluster start remote parallel ParaView server:

```
$ cd scratch/webinar   # necessary on Cedar
$ salloc --time=0:60:0 --ntasks=128 --mem-per-cpu=3600 --account=def-someuser --
    partition=cpubase_interac
$ module load gcc/9.3.0 paraview-offscreen/5.8.0
$ mpirun -np 128 pvserver
```

2. Wait for it to start waiting for incoming connection:

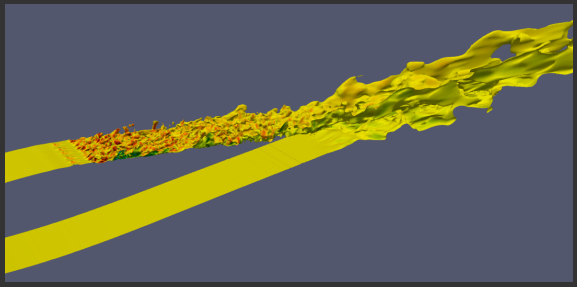
```
Waiting for client...
Connection URL: cs://cdr767.int.cedar.computecanada.ca:11111
Accepting connection(s): cdr767.int.cedar.computecanada.ca:11111
```

3. On your laptop start SSH port forwarding:

```
$ ssh cedar.computecanada.ca -L 11111:cdr767:11111   # use the actual compute node
```

4. On your laptop start ParaView 5.8.x, click Connect, then connect to `cs://localhost:11111`

5. Tools → Start Trace
6. Load OpenFOAM data, set Case Type = Decomposed
7. Apply Calculator: speed = mag(U)
8. Apply Contour at `speed=0.8`
9. Colour by $(\text{vorticity})_y$
10. Load *Rainbow Desaturated* colourmap
11. Save the image as a PNG file
12. Tools → Stop Trace
13. Save the generated script as `airflow.py` locally
    - edit it in a text editor, simplify (most lines will be setting defaults), but do not run it locally
    - provide the correct output PNG path on the remote system
    - I put the optimized version `airflow2.py` inside the ZIP file



14. Upload the script to the cluster:

```
$ scp airflow.py cedar.computecanada.ca:scratch/webinar/
```

15. On the cluster try running it as a parallel interactive job:

```
$ cd ~/scratch/webinar
$ salloc --time=0:60:0 --ntasks=128 --mem-per-cpu=3600 --account=def-someuser --
    partition=cpubase_interac
$ module load gcc/9.3.0 paraview-offscreen/5.8.0
$ mpirun -np 128 pvbatch --force-offscreen-rendering airflow.py
```

16. Once you are happy with the result, write a Slurm job submission script and submit it with `sbatch`

Remote batch animation

Option 1: (not useful for this OpenFOAM dataset) put your Python script into a loop

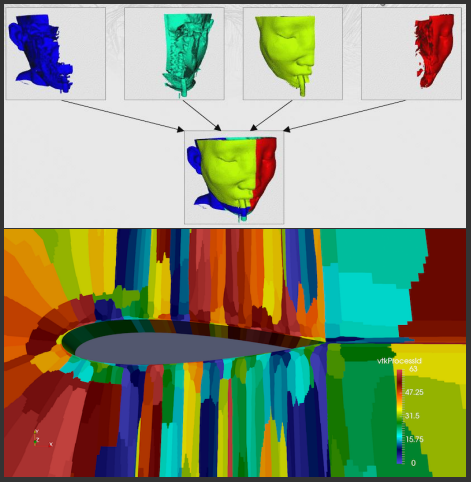
- read/write new files at the beginning/end of each loop iteration
- pay close attention to garbage collection: don't want to keep objects from previous iterations $\Rightarrow$ use `Delete()` command
- merge frames with `ffmpeg`

Option 2: (perfect for OpenFOAM) read all timesteps as a sequence/database

- instead of File $\rightarrow$ Save Screenshot... use File $\rightarrow$ Save Animation...
- important to save animation as PNG, as opposed to AVI or OGV
- script it with Tools $\rightarrow$ Start/Stop Trace
- merge frames with `ffmpeg`

Data partitioning in parallel ParaView

- If loading unpartitioned data ⇒ dynamic load balancing is handled automatically for structured data:
  - structured points
  - rectilinear grid
  - structured grid
- Unpartitioned unstructured data will usually be read in serial, then must be passed through D3 (Distributed Data Decomposition) filter for dynamic load balancing:
  - particles/unstructured points
  - polygonal data
  - unstructured grid
- Some unstructured file formats can be read in parallel, e.g. the OpenFOAM reader will automatically read its unstructured data in parallel, distributing it among all available CPU cores
- After passing your unstructured data through D3, you can save it as parallel PVTU file ⇒ you'll get a statically distributed dataset that you can load next time with the same number of CPU cores
- Reading time ⇒ you can usually tell if your dataset is being read in serial or in parallel
- Look for `vtkProcessID` variable



Data partitioning in parallel ParaView (cont.)

If you have a large (many GBs) .vtu file:

1. Read your serial .vtu file into parallel ParaView on 16 cores - slow
   - and hope that it does not run out of memory on the reading core!
   - at this point the dataset is sitting in memory on one core
   - example: serial .vtu file at 9.1GB $\Rightarrow$ 1'49" reading time
2. Apply D3 filter to distribute the dataset - slowish (memory + MPI)
3. File $\rightarrow$ Save data as .pvtu with lz4 level-6 (fast) compression - fast
   $\Rightarrow$ 16 files + 1 header file
   - now you have a statically decomposed dataset
4. Restart parallel ParaView on 16 cores, read .pvtu from scratch into - fast!
   - at this point the dataset is distributed across all 16 cores
   - example: same (but now decomposed) .pvtu dataset at 5.1GB (fast compression) $\Rightarrow$ 11" reading time

- The same I/O speeds logic applies to .vti $\rightarrow$ .pvti (but there is no need for D3)

How many CPUs/GPUs do I need?

Your main bottlenecks will be physical memory and disk read speed, and to a lesser extent CPU/GPU rendering time $\Rightarrow$ to simplify things, to decide on the number of CPU cores for initial dataset exploration, the best bet is to use the dataset size

- 128GB base node with 32 cores *minus the OS and utilities* $\Rightarrow$ **$\sim$3.5 GB/core is a good starting point** for estimating the number of cores, based on the dataset size from a single time step
- You could ask for more memory/core, but you don't want to starve other users of memory!
- Let's say, just loading your data takes $\sim$200 GB memory (single timestep!)
  - $\Rightarrow$ 58 cores
  - $\Rightarrow$ however, we also need to account for MPI buffers, filters, other data processing, possibly structured to unstructured conversion ($\sim$3X memory footprint along the interfaces)
  - $\Rightarrow$ could barely work with 64 cores
  - $\Rightarrow$ for comfortable processing with complex filters use 128 cores
- On large HPC systems ParaView is known to scale well to $\sim 10^{12}$ resolution elements on $\sim$10,000 cores and beyond
- Always do a scaling study before attempting to visualize large datasets
- It is important to understand memory requirements of filters
  - e.g. a typical (structured $\rightarrow$ unstructured) filter increases your memory footprint by $\sim$ 3X

OpenGL context for off-screen rendering on a GPU

To render on a GPU from an OpenGL application such as ParaView, traditionally you would require:

1. OpenGL support in the GPU driver, and
2. an X server that handles windows and surfaces onto which client APIs can draw
   - run X11 server (typically started by root) on the GPU compute node, set `DISPLAY=:0.\$gpuindex` (get GPU index from Slurm)

Latest NVIDIA GPU drivers include EGL (*Embedded-System Graphics Library*) support enabling creation of an OpenGL context for off-screen rendering without an X server.

- `paraview-offscreen-gpu/5.8.0` was compiled with EGL support $\Rightarrow$ use its **`pvserver`** for client-server and **`pvbatch`** for batch rendering on GPU(s) without an X server
- unlike X11, EGL does not require any special setting to scale to very high resolutions, e.g., 4K – simply set the image size to 3840×2160

GPU rendering

In all previous examples replace `paraview-offscreen/5.8.0` with `paraview-offscreen-gpu/5.8.0`, e.g.

- client-server (ParaView client must match the same major version (5.8.x))

```
$ salloc --time=0:30:0 --gres=gpu:1 --mem-per-cpu=3600 --account=def-someuser
...
$ module load gcc/9.3.0 paraview-offscreen-gpu/5.8.0
$ pvserver    # --egl-device-index=0 not needed: first GPU is #0 inside the job
```

- batch rendering

```
$ salloc --time=0:30:0 --ntasks=1 --gres=gpu:1 --mem-per-cpu=3600 --account=def-someuser
$ module load gcc/9.3.0 paraview-offscreen-gpu/5.8.0
$ pvbatch volume2.py
```

Should I use CPUs or GPUs for rendering on clusters?

- GPUs have traditionally been faster for rendering graphics ...
- In recent years better open-source software rendering libraries such as OSPRay and OpenSWR (Intel's ray tracer and rasterizer, respectively) and better OSMesa implementations have largely closed the performance gap for many types of visualizations
- Due to the rising popularity of GP-GPU computing, on Compute Canada clusters GPUs are in much higher demand than CPUs
- If you want to take advantage of parallel I/O and store large datasets in memory, you'll end up using many CPU cores anyway – why not use them for rendering?
- Having said all that, feel free to use cluster GPUs for rendering

Python scripting in VisIt

- Launching VisIt's Python scripts from the Unix command line without the GUI

```
$ /path/to/VisIt -nowin -cli -s script.py
```

  - flag `-nowin` for offscreen (typically OSMesa) rendering
  - similar to ParaView's `pvbatch`
  - very useful for running a batch rendering job on a cluster

- Launching VisIt's Python scripts from the GUI
  - VisIt has a built-in Python shell through Controls → Launch CLI... that will start VisIt's Python interpreter in a terminal and attach it to the running VisIt session on a specific port on your laptop with a one-time security key
  - alternatively, Controls → Command... provides a text editor with Python syntax highlighting and an Execute button, lets save up to eight snippets of Python code

- Recording scripts from the GUI
  - Controls → Command... window lets you record your GUI actions into Python code that you can use in your scripts, similar to ParaView's Trace Tool

Remote VisIt workflows

- For small datasets
  - use "recorder" to create a script on your laptop and debug it locally
  - copy the script to the cluster
  - run it there as a batch job
- For large datasets
  - start parallel client-server VisIt – very different from ParaView!
  - use "recorder" to save a script on your laptop, edit/clean it locally
  - copy the script to the cluster
  - run it there as a parallel batch job with `-nowin -cli -s script.py`
- Watch our "Scripting and other advanced topics in VisIt visualization" webinar from November 2016 at `http://bit.ly/vispages`
- Read `https://docs.computecanada.ca/wiki/VisIt`

Summary

- We mentioned a large array of tools: Matplotlib, Plotly, YT, ParaView, VisIt
- Any Linux visualization package with a programming interface in any language can be scripted to run on an HPC cluster
- Debug interactively locally or remotely (client-server)
- For large-scale production convert your workflow into a batch script
- We discussed CPU vs. GPU rendering + resources for large visualizations
- You can reach me via **alex.razoumov@westgrid.ca**
- Upcoming National ParaView workshops in English (October 4-5) and French (November 2-3) ➢ follow the links at `https://bit.ly/wg2021b`
- WestGrid fall webinars `https://bit.ly/wg2021b`

Questions?