Remote and distributed visualization

Visualizing remote data

Details at `https://docs.alliancecan.ca/wiki/ParaView`

If your dataset is on a remote cluster, there are several options:

✘ **download data** to your desktop and visualize it locally
- limited by the dataset size and your desktop's CPU/GPU + memory

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

✔ run ParaView remotely via **remote desktop** (via Open OnDemand or JupyterHub)
- your desktop $\xrightarrow{\text{VNC}}$ cluster running ParaView
- easiest option: use `https://jupyterhub.fir.alliancecan.ca` or similar front-end (described 2 slides from now)
- manual option: you can start a VNC server on an interactive cluster compute node by hand as described in our documentation `https://docs.alliancecan.ca/wiki/VNC`
- serial (default) or parallel (start a remote server + connect from a remote client)

✔ run ParaView in **client-server mode**
- ParaView client on your desktop $\rightleftharpoons$ serial or parallel ParaView server on larger machine

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

- render server can run with GPU rendering or purely in software
- data/render servers can run on single-core, or across several cores/nodes with MPI
- for interactive GUI work on clusters you should schedule interactive jobs, as opposed to running on the login nodes

Special remote vis cases

1. In-situ visualization = instrumenting a simulation code on the cluster to
   1.1 output graphics and/or
   1.2 act as on-the-fly server for a visualization frontend (ParaView/VisIt client on your laptop)
   - need to use a special library (ParaView's Catalyst or VisIt's libsim)
   - very advanced topic for another time
2. Web-based visualization with data served from another location
3. In May-05 webinar *"OpenStack command line in action: VMs in minutes"* I will demo:
   - creating a VM from scratch in our OpenStack cloud
   - running remote ParaView in it via VNC (serial and multi-core)
   - starting a ParaView server in it and connecting from a ParaView client on your laptop
   - ... but today is all about HPC clusters

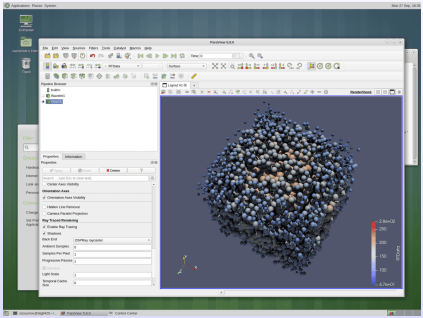
ParaView via remote desktop

Use case in `https://docs.alliancecan.ca/wiki/ParaView` | "Small-scale interactive" tab

1a. On Fir, Rorqual, or Narval: launch a JupyterLab instance
1b. On Nibi or Trillium: launch an Open OnDemand instance
   - in all cases, details at `https://docs.alliancecan.ca/wiki/systemName`

2. Start a remote desktop

3. In the terminal inside the remote desktop, load the ParaView module



4a. For serial rendering: start ParaView inside the remote desktop
4b. For multi-core rendering: `mpirun --oversubscribe -np ... pvserver`, start ParaView inside the remote desktop, connect to the local server

5. Load your data and render

Hands-on: ParaView via remote desktop

Today we'll try this on the training cluster:
`https://jupyter.cass.vastcloud.org`

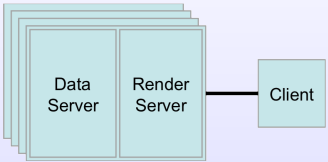
1. choose a username and get a password from the instructor
2. copy data into your remote directory, e.g.

```
$ unzip /project/def-sponsor00/shared/paraview.zip data/sineEnvelope.nc
```

3. try serial rendering
4. try parallel rendering on 4 cores + colour by ProcessID

Taking the next step: distributed client–server ParaView

Use case in `https://docs.alliancecan.ca/wiki/ParaView` | "Large-scale interactive" tab



- Local client on your computer
- Run `pvserver` on a multi-core server or a distributed cluster
- Alliance's general-purpose (variety of workloads) clusters: Fir, Nibi, Rorqual, Narval
  - located at SFU, UofWaterloo, École de technologie supérieure (Montreal)
  - specs in our documentation wiki
  - specs at `https://docs.alliancecan.ca/wiki/Fir` (replace `Fir` with `Nibi` or `Rorqual` or `Narval`)
  - batch-oriented environment for parallel and serial jobs $\Rightarrow$ use Slurm scheduler
  - identical software setup `https://docs.alliancecan.ca/wiki/Available_software`

Question 1: should I use CPUs or GPUs for rendering?

- We can render on GPUs (*hardware acceleration*) or CPUs (*software rendering*) with both interactive and batch visualization
- On desktops, graphics-focused RTX GPUs have traditionally been faster for rendering graphics
  - in a standard graphics card, a large portion of the chip is dedicated to "Graphics Processing Clusters" to run things like rasterization and texture mapping

(1) H100-class data-centre GPUs were designed for AI / HPC, not graphics
  - to lower cost, the H100 allows OpenGL/Vulkan pipelines to run on only 2 out of its ∼66 thread controllers ⇒ rendering will utilize the card only at ∼ 3% efficiency ... not the best use of this very expensive device ⇒ don't use it for visualization

(2) On clusters, far more CPU cores than GPUs ⇒ much easier to allocate many CPU cores
  - with enough CPU cores, open-source rendering libraries such as Intel's OSPRay (ray tracing) and OpenSWR / OSMesa (rasterization) have largely closed the performance gap for many visualization tasks

(3) Much faster to load *partitioned* data in parallel on 16/32/64/128 CPU cores ⇒ reuse the same cores for parallel rendering, with no data reshuffling!
  - more on I/O and data partitioning later

- Some older NVIDIA GPUs and AMD GPUs can still be used for rendering on clusters, but these are rather corner cases, so we won't focus on them here
- In this workshop, all exercises assume **CPU rendering**; slides on **GPU rendering** are also included

Question 2: how many CPUs/GPUs do I need?

- How many processors do we need? From *ParaView documentation*:
  - structured (Structured Points, Rectilinear Grid, Structured Grid): one CPU core per ∼20 million cells
  - unstructured (Unstructured Points, Polygonal Data, Unstructured Grid): one CPU core per ∼1 million cells
- Your initial bottlenecks will be physical memory and disk read speed, followed eventually by CPU/GPU rendering time $\Rightarrow$ to simplify things, to decide on the number of CPU cores for initial dataset exploration, use the dataset size
  - consider 80 GB dataset (single timestep)
  - base nodes have 128 GB memory with 32 cores $\Rightarrow$ 3.5 GB/core (accounting for the OS, system tools, etc.) $\Rightarrow$ 23 cores for this dataset
  - need to account for filters (and other processing), MPI buffers $\Rightarrow$ minimum 32 cores
  - for comfortable processing with complex filters use $48 - 64$ cores
- On large HPC clusters ParaView has been shown to scale to $\sim 10^{12}$ cells (Structured Points) on ∼10,000 cores and beyond
- Always do a scaling study before attempting to visualize large datasets
- It is important to understand the memory requirements of filters
  - a typical (structured $\rightarrow$ unstructured) filter increases memory footprint by ∼3X

Important setting: Remote Render Threshold

In ParaView's preferences can set (Render View | Remote/Parallel Rendering Options | Remote Render Threshold) beyond which rendering will be remote

- default 20MB $\Rightarrow$ small rendering will be done on your laptop's GPU, interactive rotation with a mouse will be fast, but anything modestly intensive (under 20MB) will be shipped to your laptop and might be slow
- 0MB $\Rightarrow$ all rendering (including rotation) will be remote, so you will be really using the cluster's CPU(s)/GPU(s) for everything
  - good for large data processing
  - not so good for interactivity, especially on a slower connection
- experiment with the threshold to find a suitable value

Hands-on: local client, remote software rendering

Goal: start with interactive client-server, convert to remote offscreen batch rendering

1. On the cluster start remote parallel ParaView server:

```
$ module load paraview
$ salloc --ntasks=4 --time=0:60:0 --mem-per-cpu=3600     # --account=def-someuser
$ mpirun -np 4 pvserver --force-offscreen-rendering --opengl-window-backend OSMesa
```

   - ✍ usually no need to specify the OpenGL window backend, but sometimes the default backend is not determined properly
   - ✍ how would you adapt these commands to render on one CPU?

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

```
Waiting for client...
Connection URL: cs://node1.int.cass.vastcloud.org:11111
Accepting connection(s): node1.int.cass.vastcloud.org:11111
```

3. On your computer start SSH port forwarding:

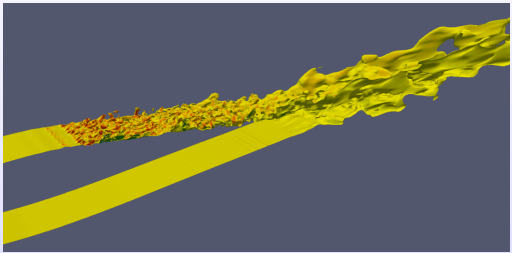
```
$ ssh userXX@cass.vastcloud.org -L 11111:node1:11111  # use the actual compute node
```

   - ✍ more details (Linux, MacOS, Windows) at `https://docs.alliancecan.ca/wiki/SSH_tunnelling`

4. On your computer start ParaView 6.0.x, click Connect, fill in the connection details, then connect to `cs://localhost:11111`

Hands-on continued

5. Tools → Start Trace
6. Load and visualize the dataset
   - use one of the four datasets described later in these slides
7. Save the image as a PNG file
8. Tools → Stop Trace
9. Save the generated script as `myscript.py` locally
   - edit it in a text editor, simplify (most generated lines will be setting defaults)
   - provide the correct input / output file paths on the remote system



Hands-on continued

10. Upload the script to the cluster:

```
$ scp myscript.py userXX@cass.vastcloud.org:
```

11. On the cluster try running it via an interactive job:

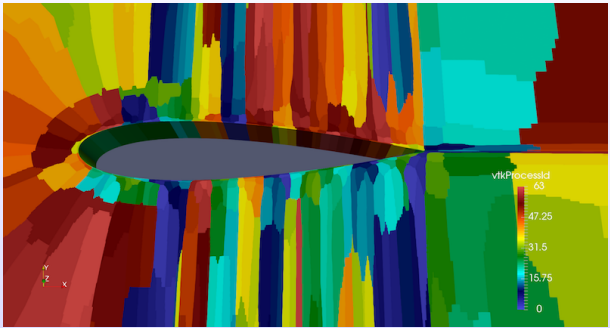
```
$ module load paraview
$ salloc --time=0:60:0 --ntasks=4 --mem-per-cpu=3600     # --account=def-someuser
$ mpirun -np 4 pvbatch --force-offscreen-rendering \
         --opengl-window-backend OSMesa myscript.py
```

    and then check the results

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

Check parallel rendering + monitor resource usage

- ParaView processes communicate via MPI
- The variable `vtkProcessID` will show you domain decomposition (more on this later)



See a more artistic version of this figure

- Check distributed memory usage: View | Memory Inspector
- Check CPU usage: `srun --jobid=<jobID> --pty htop`
- Check GPU usage: `srun --jobid=<jobID> --pty watch -n 1 nvidia-smi`

Next few pages: datasets for remote rendering exercises

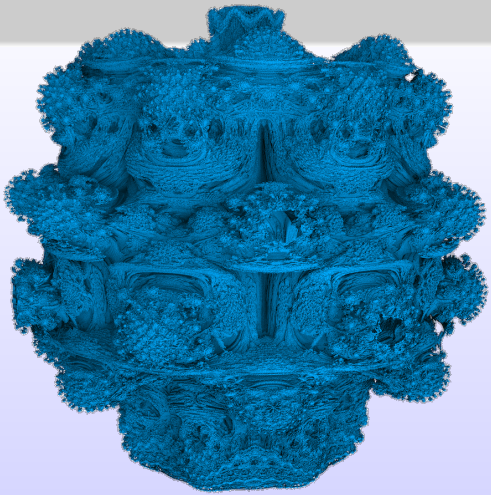
Simpler exercise:

1. Create your visualization via interactive client-server using CPU rendering
2. Save your visualization to PNG

Additional steps (extended exercise):

3. Convert this workflow into a Python script
4. Upload this Python script to the cluster
5. Try running the script inside an interactive (`salloc`) job; debug if needed
6. Once happy with the result, write a Slurm job submission script and submit this rendering as a batch (`sbatch`) job

Dataset 1: Mandelbulb



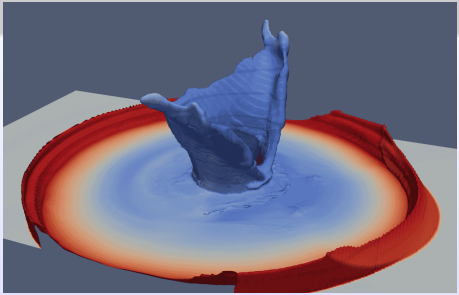
- Visualize power-8 Mandelbulb
- Use the file `mandelbulb800.nc` – sampled at 800^3
- Use 4 - 8 CPU cores on the training cluster via `salloc`
  1. try to recreate the picture on the right: pay attention to the lights and shadows
  2. use View → Memory Inspector to keep an eye on memory usage
  3. optionally colour your dataset by `processID`

- Copy the file:

```
$ unzip /project/def-sponsor00/shared/paraview.zip data/mandelbulb800.nc
$ ls -lh data/mandelbulb800.nc
```

Dataset 2: deep impact dataset

Dataset from IEEE 2018 SciVis Contest



- Dataset from *Deep Water Impact* simulation by John Patchett (LANL) and Galen Gisler (Univ. of Oslo)
  - dataset details here
  - 269 low-resolution ($460 \times 280 \times 240$) snapshots in time
  - the original simulation is much higher resolution
  - used with permission
- While you could render this dataset in serial, probably best to give it few cores
- Full dataset (115GB in total)

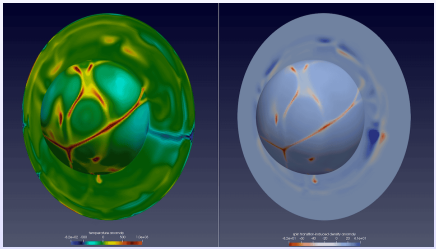
  `fir:/project/6003910/razoumov/ieeevis2018-deepWaterImpact/460x280x240`
- Reduced dataset on the training cluster `/project/def-sponsor00/shared/deepImpact`
  - every 10th file in the timeline $\Rightarrow$ 26 files, 12GB
  - use 8 cores on the training cluster
  - load all files, Contour by `v02` (water volume fraction), colour by ρ, Rescale to data range
  - alternatively, visualize `snd` (sound speed)
- To make it easier to navigate to the dataset in ParaView, create a symbolic link:

```
$ mkdir -p ~/data
$ ln -s /project/def-sponsor00/shared/deepImpact ~/data/deepImpact
```

Dataset 3: 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
  - used with permission

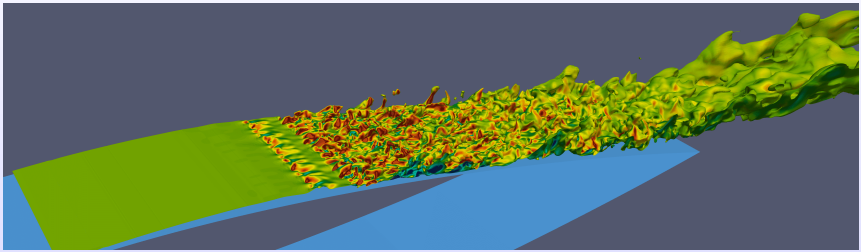


- Full dataset (89GB in total)
  `fir:/project/6003910/razoumov/ieeevis2021-mantleConvection/spherical`
- Reduced dataset on the training cluster `/project/def-sponsor00/shared/mantle`
  - every 10th file in the timeline $\Rightarrow$ 26 files, 11GB
  - you can render this dataset in serial
  - visualize any 3D scalar field, e.g. the *temperature anomaly* or the *spin transition-induced density anomaly*
- To make it easier to navigate to the dataset in ParaView, create a symbolic link:

```
$ mkdir -p ~/data
$ ln -s /project/def-sponsor00/shared/mantle ~/data/mantle
```

Dataset 4 (Fir only): airflow over a turbine blade

Dataset from WestGrid's 2019 `https://computecanada.github.io/visualizeThis` competition



- OpenFOAM *decomposed* dataset: 512 cores, 86 timesteps, 5 hydro variables, ∼1TB in total
  - simulation by Joshua Brinkerhoff (UBC Okanagan), used with permission
  - 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
  - data in `fir:/project/6003910/razoumov/visThis2019-airfoil`
- Image at the top shows the air speed isosurface coloured by the Y-component of the vorticity, full animation rendering (86 timesteps) took 17m on 128 CPU cores on Cedar (predecessor to Fir)
- Create a symbolic link to make it easier to navigate to the dataset in ParaView

Let's try animations

1. Time animation
2. Camera animation
   - check the "Scripting" section in these slides
   - suggestions: rotate 360° around the vertical axis, fly towards/through the object

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)

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

- your OpenGL application needs to be **recompiled with EGL support** $\Rightarrow$ we have built this into `paraview/6.0.0` module that provides both **`pvserver`** for client-server and **`pvbatch`** for batch rendering
- unlike X11, EGL does not require any special setting to scale to very high resolutions, e.g. 4K (3840×2160) – simply ask it to render a 4K image

Interactive client-server rendering on a cluster's GPU

1. On Narval **submit an interactive job** to the GPU partition, e.g. a serial job:

```
$ salloc --time=0:30:0 --ntasks=1 --gpus-per-node=a100:1 \
         --mem-per-cpu=3600 --account=def-someuser
```

   When the job starts, it'll return a prompt on the assigned compute node.

2. On the compute node inside the job **start the ParaView server** using a special version of ParaView with EGL support

```
$ module load paraview
$ pvserver --force-offscreen-rendering --opengl-window-backend EGL
       # --egl-device-index=0 not needed: first available GPU inside the job is 0
```

   For multiple GPUs can use

```
$ nvidia-smi -L    # will return 0, 1, ...
```

   The `pvserver` command will return something like

```
Waiting for client...
Connection URL: cs://ng20201:11111
Accepting connection(s): ng20201:11111
```

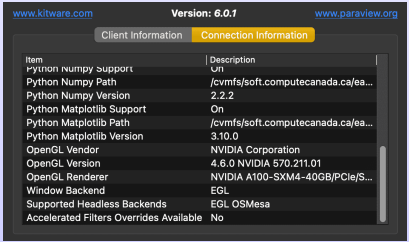
Interactive client-server rendering on a cluster's GPU (cont.)

3. On your desktop **set up ssh forwarding** to the ParaView server port:

```
$ ssh username@narval.alliancecan.ca -L 11111:ng20201:11111
```

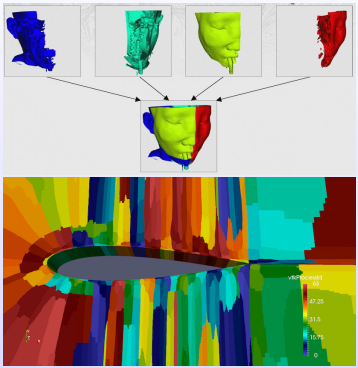
4. On your computer **start ParaView 6.0.x** and **edit its connection properties** under *File - Connect - Add Server* (name = local11111, server type = Client/Server, host = localhost, port = 11111), click *Configure* → *Manual* → *Save*, then select the server from the list and click on *Connect*
5. Remote rendering should be done on the GPU

- To confirm, select ParaView | About ParaView | click on the Connection Information tab and check under OpenGL Renderer
- For a successful client–server handshake, ParaView's client and server must have matching major versions (6.0)



Data partitioning in parallel ParaView

- If loading unpartitioned structured data ⇒ dynamic load balancing is handled automatically
  - 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
- Partitioned unstructured data can be read in parallel, e.g. the OpenFOAM reader will automatically read its decomposed data in parallel, distributing it among all available CPU cores



- After passing your unstructured data through D3, you can save it as a 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
- With data in memory, look for `vtkProcessID` variable

Data partitioning in parallel ParaView (cont.)

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

1. Read this file in 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 no need for D3)

Exercise: parallel rendering of partitioned data

This is a step-by-step guide for an unpartitioned, multi-block turbine dataset in `/project/def-sponsor00/shared/originalNonDecomposed` (∼18GB)

```
$ ln -s /project/def-sponsor00/shared/originalNonDecomposed ~/data/originalNonDecomposed
```

1. Partition the data
   - 1.1 `salloc --time=0:60:0 --ntasks=16 --mem-per-cpu=3600`
   - 1.2 start client-server ParaView session on 16 cores
   - 1.3 load all `.vtm` files (all 10 timesteps); each points to a subdirectory with `.vtu` files
   - 1.4 apply Merge Blocks, output type = Unstructured Grid
   - 1.5 apply Cell Data to Point Data (so that you could use Contour)
   - 1.6 apply D3
   - 1.7 save data as `decomposed.pvtu`, write all timesteps as a series, use fast compression
     ⌛ this last step might take a while (10-15 mins) for all 10 timesteps; what's in the files now?
2. Visualize it
   - 2.1 reset your client-server ParaView session on 16 cores ⌛ to test the reading speed
   - 2.2 load all `decomposed.pvtu` files ⌛ do they load faster?
   - 2.3 create visualization interactively, save animation as 1000 × 800 PNG files ✍ ∼1 min
   - 2.4 merge them into a movie with `ffmpeg`

Working with large datasets

Some filters **should not be used with structured data**:
they write unstructured data, can be heavy on memory usage

- Append Datasets
- Append Geometry
- Clean
- Clean to Grid
- Connectivity
- D3
- Delaunay 2D/3D
- Extract Edges
- Linear Extrusion
- Loop Subdivision
- Reflect
- Rotational Extrusion
- Shrink
- Smooth
- Subdivide
- Tessellate
- Tetrahedralize
- Triangle Strips
- Triangulate

Use these with caution: Clip, Decimate, Extract Cells by Region, Extract Selection, Quadric Clustering, Threshold (also write unstructured, but not as heavy on memory)

Remote rendering: key orthogonal decisions

(1) interactive vs. batch

- Interactive client-server for a quick look, exploration or debugging
  - another option is to download a scaled-down version of your dataset, debug a script locally on your laptop, and then run it as a batch job on the original full-resolution dataset on the cluster
- Batch really preferred for production jobs and producing animations

(2) CPU vs. GPU

- On clusters, CPU rendering is typically the best choice
  - recall: data-centre GPUs perform poorly for graphics workloads
  - you can throw many CPUs at your rendering job
  - modern ray tracing and rasterization libraries can be very fast at scale
  - I/O benefits
- For initial exploration, use the dataset size (in GB) to estimate the appropriate number of CPU cores, then adjust as needed