2022 Student Summer Internship: Visualisation of Large Cumulus Clouds Datasets

Documentation

September 23, 2022

Contents

1 Summary

Work focused on visualisation of detailed simulations of cumulus clouds performed on supercomputing systems, including `ARCHER2` at `EPCC`.

Initial work concentrated on the requirements analysis for ensuring scalable visualisation for datasets of tens to hundreds of gigabytes (`GB`) per time step. Analysis included a detailed examination of the current state of the art in visualisation and how it worked on a large scale. This was followed by research and set-up of `ParaView` on supercomputing systems.

Finally, the project looked at ways to maximise the dissemination of the project. Final outcome includes realistic visualisations to the public at science festivals, systematic generation of videos showing the development of a cloud, scientific-analysis and the documentation of the set-up and operation of the visualisation on the supercomputing systems.

2 Deliverables

- Repository with automation scripts and documentation (a)
- Requirements Analysis Document (b)
- ParaView User's Guide: The Summary (c)
- Pipeline Documentation (this document) (d)
- Cluster Performance Analysis (e)
- Cluster Connecting and File Transfer Commands (f)

3 Proposed Visualisation Pipeline

This section describes a suggested pipeline, consisting of the two identified stages:
Stage 1: Set-up of the Visualisation and Analysis Environment
Stage 2: Automation of the Visualisation Pipeline

Both stages utilise existing modes of operations on supercomputing clusters - called *interactive* (useful for Stage 1) and *batch* (for Stage 2).

3.1 Stage 1: Visualisation Set-up

This stage involves data analysis. As the final result is a rendered view, either in image, or video form, the scientist needs to determine the most suitable camera view and visualisation settings for the purpose.

For this, the most appropriate cluster-usage method was determined to be *interactive* mode

- allowing interacting with `ParaView` in real-time. On supercomputer clusters, the use of remote *interactive* mode allows rendering on the server (the cluster) and launching `ParaView` Graphical User Interface (GUI) on the client, where the user can tweak relevant settings live, before setting up any automated scripts. The Render-view window is replaced by images that the server sends back to the client.

Next, we describe ways to interactively prepare the pipeline for large-scale batch submissions. Note, that for small data, it might be possible to produce decent output in the interactive mode alone. The next sections are more applicable to large data visualisations.

3.1.1 Loading the data in

This project and code/scripts associated, assume `NetCDF` files. One such file is chosen with `File->Open`, a pop-up menu shows up labelled *"Open Data With..."* - the default `NetCDF Reader` should work for most purposes.

Once the reader is chosen, the pop-up disappears but `ParaView` will not start reading the data in until the green *Apply* button is not clicked (if the button cannot be seen, enable Properties with `View->Properties`). Then, if the simulation is on a regular grid, the *Spherical Coordinates* checkbox can be toggled off, and the green *Apply* button can finally be clicked.

Depending on the data size, `ParaView` will either display an outline immediately, or take some time to read the data, and then display the outline once it is done.

To change rendering mode from *Outline*, in *Properties* window (the same window mentioned in the earlier paragraph), go to *Display* section, and switch *Representation Outline* to *Representation Volume*, or other. The main variable that the colour map is based on is chosen in the *Coloring* section, just under the *Representation*.

3.1.2 Method 1: General State-file Visualisations

For most general visualisations, where the interactive part is setting up how the final view will look like, the easiest identified approach was *state files*.

`ParaView` state files (`.pvsm`) are text(markup) or python (`.py`) files that save the entire application state, including any applied filters, colours, transfer functions and data sources[1].

Regarding the accuracy, markup `.pvsm` file are more reliable, as they operate in *"what you see is what you get"* mode, meaning, that the state saved will look identical when loaded next time. This is not guaranteed when using `python` statefiles, however, with some manual tweaking, it is possible. The downside of regular `.pvsm` files is that they are not as easily editable, however, being `.xml` text files, they are perfect if the only variable is the data source file, as that can be changed with find/replace functionality on any text editor.

State files can be loaded from inside `ParaView` (`File->Load State`). This makes it very convenient to load in a visualisation file that needs to be tweaked, or by saving some unfinished state progress to continue working on later.

This covers the first part of *Stage 1* - Visualisation Setup, but it is not enough for batch submissions yet. Next part (Subsection 3.1.3) will cover an alternative way to set-up the visualisation interactively, and then finally Section 3.2 will demonstrate how to combine the interactively generated state files or scripts with batch automation.

3.1.3 Method 2: Trace-file Generation

In addition to state files described in part one (Subsection 3.1.2), the user can trace `python` script files (`Tools->Start Trace`). After starting the trace, the user performs a sequence of actions that will result the desired output image/rendering* and Stops the trace (`Tools->Stop Trace`). Then, a generated `python` script is presented to the user with all performed actions translated into code.

*Note, that by generating the trace, if any video/image output is desired, then the user must explicitly save an image/video animation *during* the tracing procedure. Otherwise, the code relevant to saving can be added later.

Unlike markup `.pvsm` state files, trace-generated scripts are ready for automation (Stage 2). However, one must be careful to make sure the file paths are accessible, if this script is to be used as a batch job executable. Therefore, a good idea is to first test it by running in manual mode (either by `pvpython` ... `>>> pvpython import yourscriptname`, or `pvbatch yourscriptname.py`).

Worth noting is that this approach is quite specific, and works only with data specified within the file. To help make this format better, custom scripts made as part of the 2022 Summer Visualisation Internship will be introduced in the next section 3.2. In general, those custom scripts are general versions of traced-generated scripts, that allow plenty of command-line arguments relating to data and visualisation.

3.2 Automation

This section describes the whole automation process, first by turning a state file (from Method 1 of Visualisation set-up) into a usable script, then by running it as a batch script (or testing it locally).

3.2.1 Turning State-file into executable

A simple markup state-file on its own can only be used inside `ParaView` in interactive mode, when loading it in. However, to execute it automatically, we need to load it in a script file. This is exactly what the custom-made `python` executable
`ParaViewCloudVis/automation_scripts/cloudvis_statefile_automation.py` does.

Details on the usage are found in the `GitHub` repository `README.md` documentation, however, the key idea is to *"wrap around"* the state file - load it, set-up the camera position, lighting position, and then save the output in a chosen format (image/video or ray-traced image/video) [and optionally, to apply the cross-section filter on a chosen plane].

3.2.2 Skipping State-files / premade scripts altogether

If a fairly general visualisation is required, we can skip state files as input, and instead, provide the data source file and a transfer function.

Script, prepared for this can be found in:
`ParaViewCloudVis/automation_scripts/cloudvis_pipeline_automation.py`.

This script essentially sets everything from scratch - loads in specified data, colours it with specified transfer function, sets up the camera, lighting and saves the output in a chosen format (image/video or ray-traced image/video) [and optionally applies the cross-section filter on chosen plane].

*Notice, that this script does everything that the state-file provides, however here the only difference is that we provide the data file and transfer function instead of a state file.

Regarding tranfser files, a `.json` colour-opacity mapping file, an example is provided in
`ParaViewCloudVis/transfer_functions/hl_Cloud_white_realistic_tf.json`,
and they can be generated by the user in `ParaView` GUI, following instructions shown in *Figure 1.*

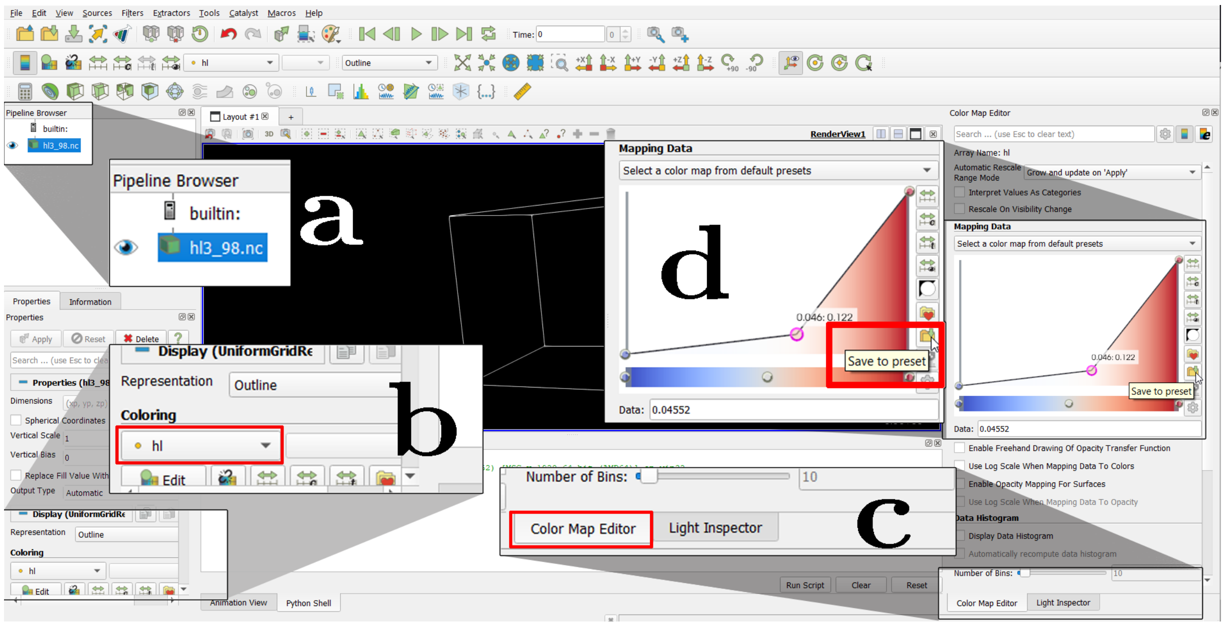


Figure 1: How to save custom transfer functions. Note, make sure there is a loaded dataset (a), the colouring field is set to a variable that has a valid range, such as 'hl' (b), then making sure that *Color Map Editor* Window is opened (c), and finally by clicking *Save as preset* button (d). The transfer function can be exported as `json` file by following the *Save as preset* menu wizard through.

Also note, that the automation scripts require not only the *path* to the transfer function, but also its *name*. This name can be found inside the `.json` file as the `"Name"`, for example: `"Name" : "WhiteCloud"`

After choosing one of the methods described, we either have a `ParaView Python (pvpython) Script`, or a `ParaView Python Batch (pvbatch) Script`.
Key distinction here in the naming is purely based on how we tend to use that script:

- `pvpython`: If the script is used inside of `ParaView` GUI (`View->Python Shell->Run Script`), or run from within `pvpython` (as: `import scriptname`)
- `pvbatch`: If the script was run as stand-alone (`python scriptname.py`), or as part of a job submission on a cluster `srun --distribution=block:block --hint=nomultithread pvbatch scriptname.py`

A good idea is to test the script with the chosen settings as a `pvpython` script, or running it as stand-alone `pvbatch` mode with `python scriptname.py` and making sure the result is as expected (for a smaller problem), before submitting it for a bigger problem.

4 Batch Submission

For the scope of this project, only the batch submission on SLURM Load Manager systems was set-up. Orignally following the documentation on ARCHER2[7] and ParaView batch programming did not work, therefore, after following the more general batch submission documentation on ARCHER2 it was updated[7](b) to:

```
#!/bin/bash
# Slurm job options (job-name, compute nodes, job time)
#SBATCH --job-name=Example_MPI_Job
#SBATCH --time=0:20:0
#SBATCH --nodes=1
#SBATCH --tasks-per-node=32
#SBATCH --cpus-per-task=4

# Replace [budget code] below with your budget code (e.g.  t01)
#SBATCH --account=[budget code]
#SBATCH --partition=highmem
#SBATCH --qos=highmem

# Set the number of threads to 1
# This prevents any threaded system libraries from automatically
# using threading.
export OMP_NUM_THREADS=1

# Launch the parallel job
# Using 32*1 MPI processes and 32 MPI processes per node
# srun picks up the distribution from the sbatch options


module load epcc-setup-env
module load load-epcc-module


module load paraview/5.10.1


srun --distribution=block:block --hint=nomultithread pvbatch pvbatchscript.py
```

Note that SLURM Load Manager supports array jobs, which can be set-up by adding:
`#SBATCH --array=0-15:4` to the batch script above,
where '0-15' are the job IDs and :4 specifies the stride (so in this case 4 jobs would be created,

as 16 divided by the stride). This is particularly useful when rendering video as a series of `.png` images, where we can set the start frame / end frame automatically, as each image is independent:

```
FRAME_START = int(os.getenv('SLURM_ARRAY_TASK_ID' , 10))
FRAME_END = int(os.getenv('SLURM_ARRAY_TASK_STEP' , 10))
```

This is explained in detail in the KAUST Visualisation Core Lab Advanced Scientific Visualization Workflows with ParaView (Spring 2021) Tutorial [2].

Note this, and other scripts can be found in the Repository, which is **Deliverable (a)**.

5 Parallel Client-Server Rendering / Connecting

Since data to be rendered exceed sizes that can be worked with on personal computers, the visualisation relies on the client-server rendering that `ParaView` provides. This section will go over the general way of connecting to a server renderer in parallel, and using the GUI from the client (when server sends back rendered results as images to the client seamlessly, replacing the usual viewport).

> Note that parallel and remote are separate, and it is possible to use either parallel or remote exclusively, but in our case, we use both parallel and remote

Summarised here are steps of running the `pvserver` in parallel and connecting to it from the client in load managers `Sun Grid Engine` (SGE) / `Son of Grid Engine` (used by `ARC3`/`ARC4`) and `SLURM` (used by `ARCHER2`).

Connecting to clusters and running the software is discussed in detail in **Deliverable (f)**.

5.1 `Sun Grid Engine` (SGE) / `Son of Grid Engine` - `ARC3/4`

Note, for the reader's knowledge, we first show how a **non**-parallel request looks like, so that it is not confused with the parallel request:

- Adding the module (requires `mesa`: `module add mesa paraview`)
- Requesting an interactive node/memory:
  whole node:
  `qrsh -cwd -V -l h_rt=00:20:00 -l nodes=1 pvserver --server-port=11111 &`
  10GB memory:
  `qrsh -cwd -V -l h_rt=00:20:00 -l h_vmem=10G pvserver --server-port=11111`

Note, that `qrsh` corresponds to an *interactive* job submission (where the user can remotely run `ParaView`, and interact with the application live via client GUI.

`-cwd` means using the current working directory.

`h_rt` is the time the request is for (hh:mm:ss format). `node/h_vmem` is either the number of nodes, or memory requested. `pvserver` is the application we are launching.

`server-port=11111` will be used for the client connection in the next step. Number `11111` is the default port (if the port is busy, this number can be freely changed).

In Parallel, this request looks as following:

```
qrsh -l h_rt=1:00:00,h_vmem=16G -pe smp 8
```

Where we no longer request an entire node, but rather memory, with the `smp N` flag (where `N` is the number of `MPI` processes, in this case shown above - 8). To connect, a new terminal session needs to be openedd (as a new window etc.) and we use `ssh-tunnelling` for setting up the connection:

```
ssh -L 11111:givennodeID:11111 login1.arc4.leeds.ac.uk
```

where the given node ID can be seen in the connection URL after running the `pvserver` with the previous command.

Finally, the same version of `ParaView` as the version on the cluster needs to be opened locally. Connection is finished with `File->Connect`. Details are found in the `ARC3` and `ARC4` documentation.

5.2 `SLURM - ARCHER2`

An interactive node is requested first:

```
srun --nodes=1 --exclusive --time=00:20:00
--partition=standard --qos=short --reservation=shortqos --pty /bin/bash
```

And after (if) it has successfully been allocated, we run paraview with:

```
cd /work/.../ module load paraview/5.10.1 srun --oversubscribe -n 64 pvserver
--mpi --force-offscreen-rendering
```

Where `-n` is the number of MPI processes.

The connection follows same procedure as in section above about `SGE, ARC` machines. Detailed description of this procedure is found on the `ARCHER2, ParaView` documentation.

6 Additional Notes

6.1 Scalability

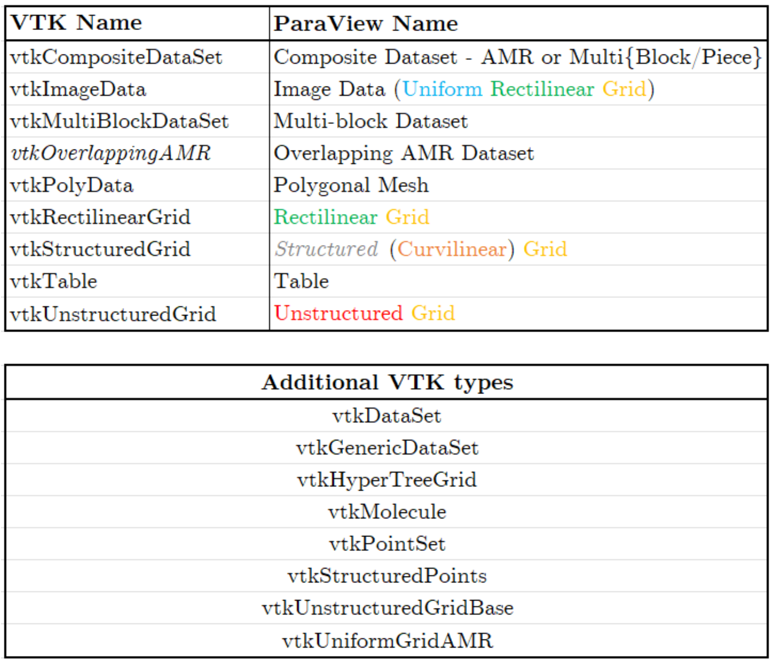
If the problem size is fixed, adding more processors does not scale quick (see Amdahl's and Gustafson's Law). On ParaView, scaling is more pronounced as data sizes grow.

6.2 Data Types

For Volume Rendering, the most efficient data format is the *Image/Uniform Rectilinear Grid*, which is luckily the format that most simulation data is already in (the gridded datasets).

However, before discussing filters in next section (6.3 Notes on Filters), we introduce the reader to available data types and their names (which might get confusing, because `ParaView` uses different names than the `VTK` framework which is at the backend of `ParaView`.

The `VTK` and `ParaView` equivalent types are shown in *Figure 2*



VTK Name	ParaView Name
vtkCompositeDataSet	Composite Dataset - AMR or Multi{Block/Piece}
vtkImageData	Image Data (Uniform Rectilinear Grid)
vtkMultiBlockDataSet	Multi-block Dataset
vtkOverlappingAMR	Overlapping AMR Dataset
vtkPolyData	Polygonal Mesh
vtkRectilinearGrid	Rectilinear Grid
vtkStructuredGrid	*Structured* (Curvilinear) Grid
vtkTable	Table
vtkUnstructuredGrid	Unstructured Grid

Additional VTK types
vtkDataSet
vtkGenericDataSet
vtkHyperTreeGrid
vtkMolecule
vtkPointSet
vtkStructuredPoints
vtkUnstructuredGridBase
vtkUniformGridAMR

Figure 2: `VTK` and `ParaView` type equivalents, as well as other `VTK` types

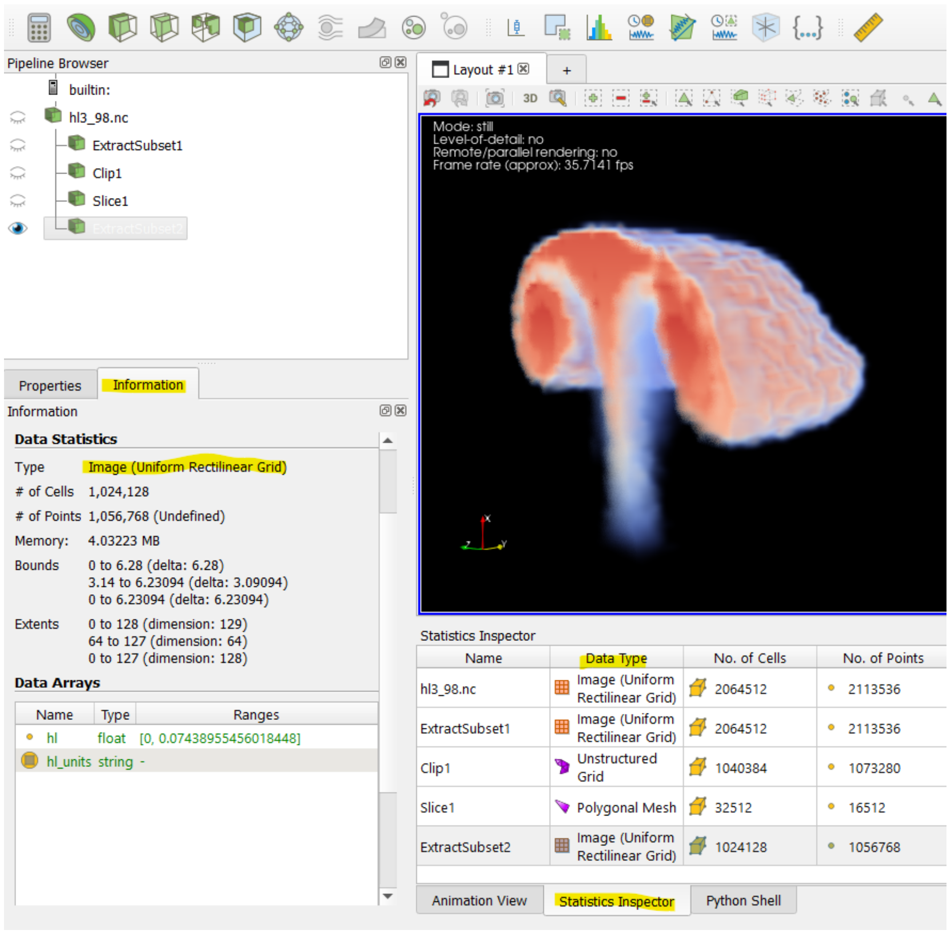
Different filters might only be available for one/some of these types.

To see what the dataset type is, there are two ways:
- Information menu, next to `Properties` - can be turned on `View->Information`.
- Statistics Inspector, at the bottom - can be turned on by `View->Statistics Inspector`.

They are shown in *Figure 3*.

6.3 Notes on Filters

Filters are a key part of `ParaView` data analysis. There are 215 Filters as of 2022 `ParaView` version `5.10.1`, which are described in detail in the official documentation[5]. That is too



Information

Data Statistics

Type	Image (Uniform Rectilinear Grid)
# of Cells	1,024,128
# of Points	1,056,768 (Undefined)
Memory:	4.03223 MB
Bounds	0 to 6.28 (delta: 6.28) 3.14 to 6.23094 (delta: 3.09094) 0 to 6.23094 (delta: 6.23094)
Extents	0 to 128 (dimension: 129) 64 to 127 (dimension: 64) 0 to 127 (dimension: 128)

Data Arrays

Name	Type	Ranges
hl	float	[0, 0.07438955456018448]
hl_units	string	-

Statistics Inspector

Name	Data Type	No. of Cells	No. of Points
hl3_98.nc	Image (Uniform Rectilinear Grid)	2064512	2113536
ExtractSubset1	Image (Uniform Rectilinear Grid)	2064512	2113536
Clip1	Unstructured Grid	1040384	1073280
Slice1	Polygonal Mesh	32512	16512
ExtractSubset2	Image (Uniform Rectilinear Grid)	1024128	1056768

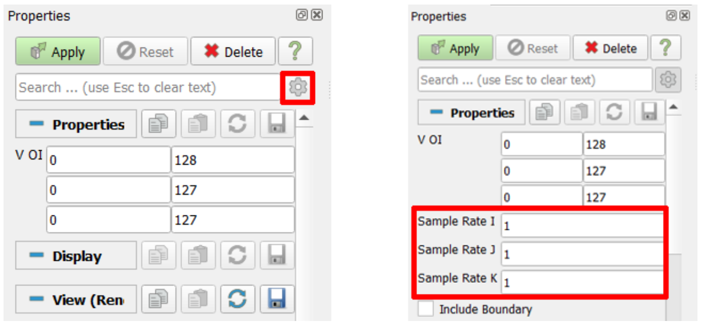
Figure 3: Two ways to see the data type. Notice, that *Statistics Inspector* shows some additional useful information.

many to consider here, but this subsection will document the filters most relevant to the cloud visualisation pipeline, as well as note that filters might change the data type, which can make rendering much slower.

6.3.1 Extract Subset

When working with large data, it is useful to set-up the visualisation using a smaller resolution version of the same dataset (especially testing during interactive mode), and then scale back to full resolution. This is possible using the *Extract Subset* filter.
Note, that the primary usage of this filter is cutting the data - extracting subsets of the data. To access the *sampling* options, it is necessary to enable *advanced options*, shown as the *gear button* (Figure 4).

Note, that the *gear button* always indicates **advanced settings/options**, and it is very easy to miss settings when following tutorials/documentation online, so make sure you are well-aware of this *advanced settings button*.



V OI 0 128
0 127
0 127

V OI 0 128
0 127
0 127
Sample Rate I 1
Sample Rate J 1
Sample Rate K 1

Figure 4: By clicking the *gear* button, we can enable sampling options (in x, y and z directions, here labelled i, j and k)

This allows down-sampling original data, at different rates in each direction.

6.3.2 Histograms

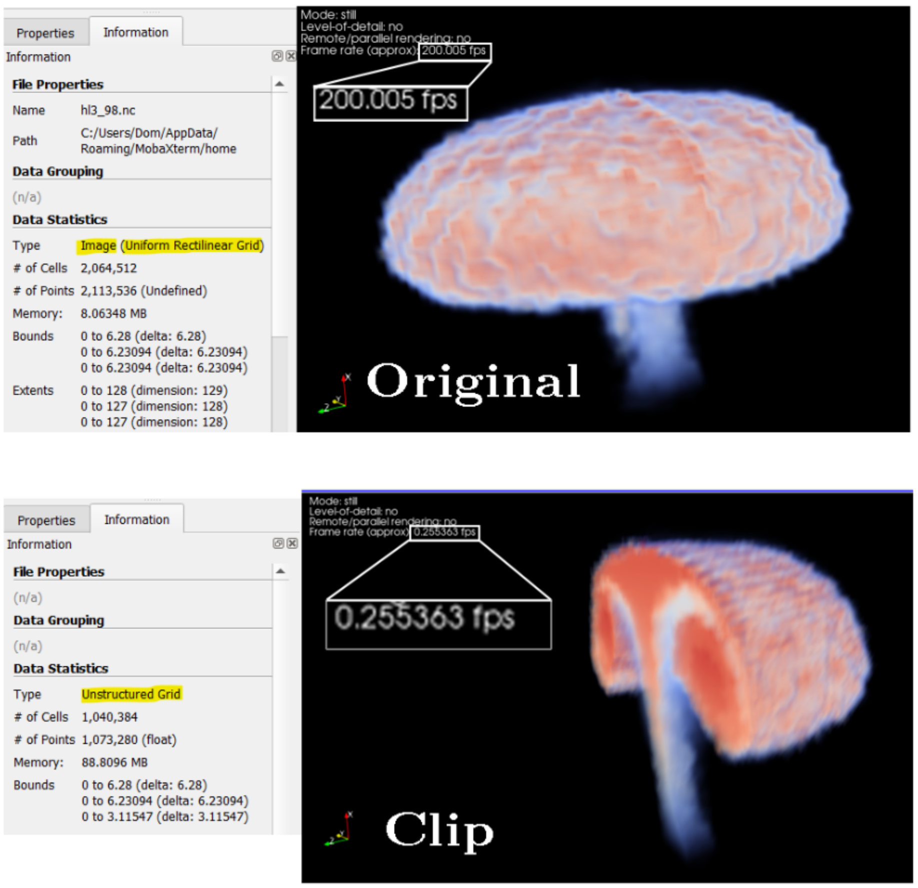
For quick data analysis on potentially interesting feature detection, a *Histogram* is a reasonable tool. It is accessible from `Filters->Data Analysis->Histogram`.

Note, that on `ARCHER2`, an alternative is available in the form of a Continuous Scatter Plot[6]. Its usage is somewhat advanced, however, worth exploring for the future data analysis. How to run `ParaView` with `TTK` and `VTK-m` is explained in **Deliverable (f)**.

6.3.3 Cross-sections

There are two original cross-section filter in `ParaView` - `Clip` and `Slice`.

This is where the user should be **warned** - **Clip** is a wasteful filter to use in this case, since it transforms the data from *Image/Uniform Rectilinear Grid* to *Unstructured Grid*, which has significant performance loss for Volume Rendering - about 800 times slower for the `128x128x128` example shown in *Figure 5*.
This lack of performance does not mean we cannot get cross-sections. An alternative approach is the *Slice* filter mentioned before, which is shown in *Figure 6*.



Information

File Properties

Name	hl3_98.nc
Path	C:/Users/Dom/AppData/Roaming/MobaXterm/home

Data Grouping

(n/a)

Data Statistics

Type	Image (Uniform Rectilinear Grid)
# of Cells	2,064,512
# of Points	2,113,536 (Undefined)
Memory:	8.06348 MB
Bounds	0 to 6.28 (delta: 6.28) 0 to 6.23094 (delta: 6.23094) 0 to 6.23094 (delta: 6.23094)
Extents	0 to 128 (dimension: 129) 0 to 127 (dimension: 128) 0 to 127 (dimension: 128)

```
Mode: still
Level-of-detail: no
Remote/parallel rendering: no
Frame rate (approx): 200.005 fps
```

200.005 fps

Information

File Properties

(n/a)

Data Grouping

(n/a)

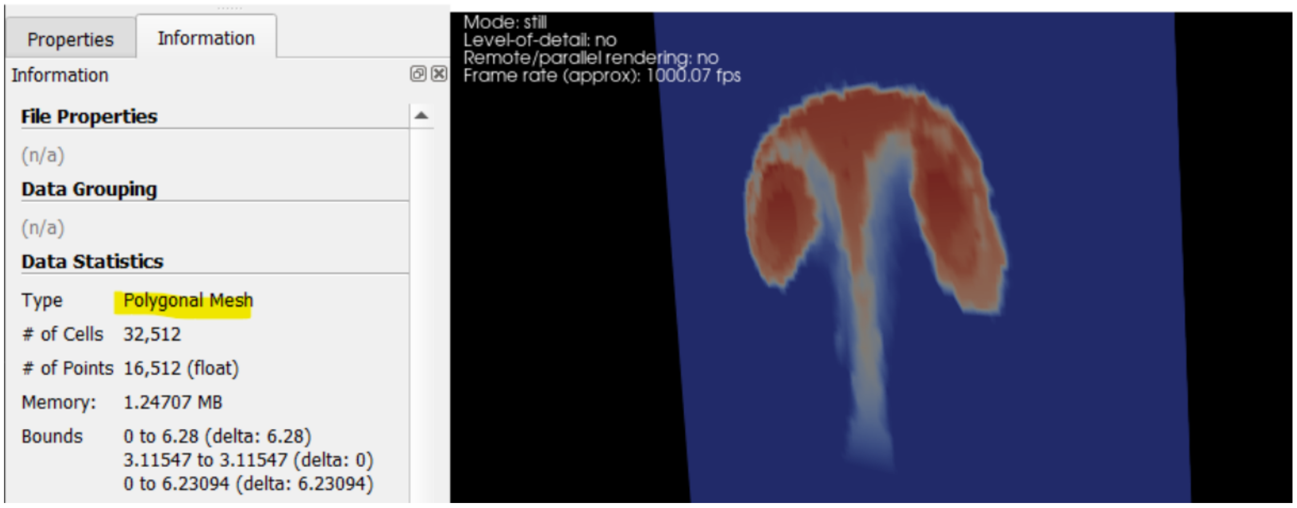
Data Statistics

Type	Unstructured Grid
# of Cells	1,040,384
# of Points	1,073,280 (float)
Memory:	88.8096 MB
Bounds	0 to 6.28 (delta: 6.28) 0 to 6.23094 (delta: 6.23094) 0 to 3.11547 (delta: 3.11547)

```
Mode: still
Level-of-detail: no
Remote/parallel rendering: no
Frame rate (approx): 0.255363 fps
```

0.255363 fps

Figure 5: By choosing the clip filter, we lose significant performance, therefore, it should be avoided



Information

File Properties

(n/a)

Data Grouping

(n/a)

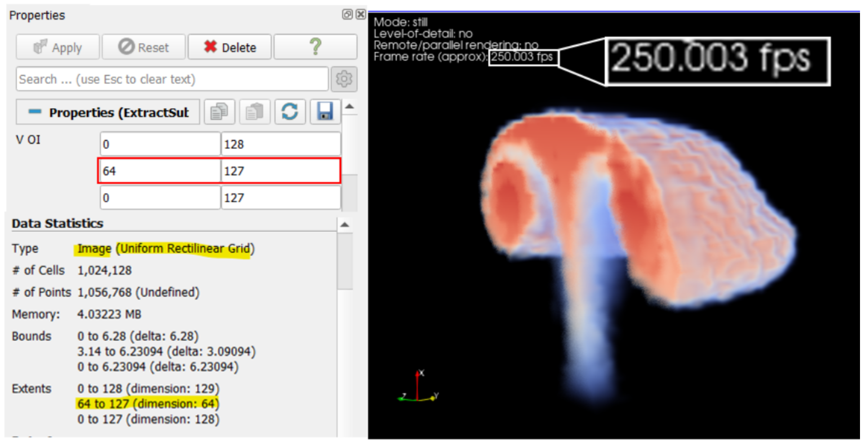
Data Statistics

Type	Polygonal Mesh
# of Cells	32,512
# of Points	16,512 (float)
Memory:	1.24707 MB
Bounds	0 to 6.28 (delta: 6.28) 3.11547 to 3.11547 (delta: 0) 0 to 6.23094 (delta: 6.23094)

```
Mode: still
Level-of-detail: no
Remote/parallel rendering: no
Frame rate (approx): 1000.07 fps
```

Figure 6: By choosing the clip filter, the data is converted to a polygonal mesh slice, which runs fast, but we lose the volumetric visualisation capability.

It still seems like we can do much better. The reader might remember the filter discussed in the previous subsection (6.3.1 Extract Subset), and that among its subsampling capabilities, the main purpose is to extract a subset of the original data. Good news is that this filter does not change the data format, so we can use it in a smart way (asking it to give us half the data) and get a result identical to *Figure 5*, with great performance (see *Figure 7*).



V OI	0	128
	64	127
	0	127

Data Statistics

Type	Image (Uniform Rectilinear Grid)
# of Cells	1,024,128
# of Points	1,056,768 (Undefined)
Memory:	4.03223 MB
Bounds	0 to 6.28 (delta: 6.28) 3.14 to 6.23094 (delta: 3.09094) 0 to 6.23094 (delta: 6.23094)
Extents	0 to 128 (dimension: 129) 64 to 127 (dimension: 64) 0 to 127 (dimension: 128)

```
Mode: still
Level-of-detail: no
Remote/parallel rendering: no
Frame rate (approx): 250.003 fps
```

250.003 fps

Figure 7: By using the *Extract Subset* filter in a smart way, we can immitate what the *Clip* filter does, just in a more efficient way, as the data type remains Image/Uniform Rectilinear Grid, keeping framerate the same.

This is not difficult to set up manually, however, since there is no direct filter which does this, cross-section building tool has been included in all automation scripts. The relevant flags to use with scripts are:
`--crossect [APPLY_CROSS_SECTION]`

`--crossect_plane [CROSS_SECTION_PLANE]`
`--crossect_percent [CROSS_SECTION_PERCENTAGE]`,
which, in this order: apply the *Extract Subset* filter, select which axis (x, y or z) to cut, and lastly select the percentage to cut. The documentation in `README.md` provides more details on the usage.

6.3.4 Iso-surfaces

There are three known ways to extract iso-surfaces:

1. (With Advanced Settings on - *gear button* pressed):
   `Properties->Display->Representation->Volume`,
   ... then scrolling down to:
   `Display->Composite`
   ... and changing it to `Isosurface`.

   Then the user needs to select a value from which to generate surfaces.
   This was has been found the least stable (sometimes crashing)**Note** - this is not a filter, but the remaining two methods are.

2. Filter *Threshold* - this will allow picking a lower and upper threshold, and convert values below/above/between that threshold into a surface. However, note, that this either uses *cuberilles* or *nearest-neighbour* interpolation, as the rendered result looks blocky.

3. Filter *Contour*. This allows specifying a range of values to generate contours from, and can be made semi-transparent by the *Enable Opacity Mapping For Surfaces* checkbox in the Colour Mapping panel. This produces a smooth surface.

From the results during this project, ***Contour*** produces the best looking surface.

6.4 Singularity Containers

Some modules might not be accessible on `ARCHER2` - Singularity containers help with that[3]. Following [3], a container for `ImageMagick` was set-up for archer and can be found in:
`/work/e710/shared/singularity/display.simg`

Similar `singularity` container is also used in the `ParaView Superbuild` to run `VTK-m` and `TTK`, which is described in **Deliverable (f)**.

Note, that work directory is not accessable by `singularity`, therefore, before viewing/working with images, they must be moved to `home` directory.

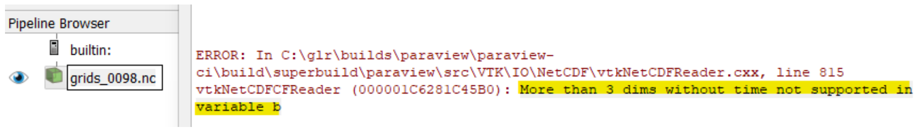
6.5 Gridded Data - removing excess variables

Gridded files might be big, but not necessarily because of the resolution, rather because of the number of extra variables included. To remove/keep certain variables, we use the `nco` module[4] (`module loadnco`):

```
ncks -v variable_name_to_keep filename.nc output.nc
```

To remove the time variable (which `ParaView` does not like, as it thinks it is the fourth dimension - *Figure 8*), we use:

```
ncwa -a time original.nc output.nc
```



```
ERROR: In C:\glr\builds\paraview\paraview-
ci\build\superbuild\paraview\src\VTK\IO\NetCDF\vtkNetCDFReader.cxx, line 815
vtkNetCDFCFReader (000001C6281C45B0): More than 3 dims without time not supported in
variable b
```

Figure 8: `ParaView` Error, when *"time"* exists as both a dimension and a variable; since *"time"* is just a label, it is interpreted as a 4th spacial dimension (as in `x, y, z, w`)

References

[1] Kitware Public *Advanced State Management*. 10 November 2020
`https://www.paraview.org/Wiki/Advanced_State_Management`

[2] KAUST Visualization Core Lab *Advanced Scientific Visualization Workflows with ParaView*. Spring 2021
https://www.youtube.com/watch?v=yhsTavGTVt4t=6699s

[3] David Henty EPCC *Using Containers to Install GUI-based Tools on ARCHER2*. September 21, 2021
https://www.archer2.ac.uk/news/2021/09/21/container.html

[4] nco(1) - Linux man page https://linux.die.net/man/1/ncwa

[5] ParaView Developers *ParaView User's Guide 5. Filtering Data*. 2020
https://docs.paraview.org/en/latest/UsersGuide/filteringData.html

[6] Bachthaler, Sven Weiskopf, Daniel. (2008). Continuous Scatterplots. IEEE transactions on visualization and computer graphics. 14. 1428-35. 10.1109/TVCG.2008.119.

[7] ARCHER2 UK National Supercomputing Service *Example: job submission script for MPI parallel job*. No date
(a) https://docs.archer2.ac.uk/user-guide/scheduler/
ParaView — Using batch-mode (pvbatch)
(b) https://docs.archer2.ac.uk/data-tools/paraview/