2022 Student Summer Internship: Visualisation of Large Cumulus Clouds Datasets

Documentation

September 20, 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
- Requirements Analysis Document
- ParaView User's Guide: The Summary

3 Proposed Visualisation Pipeline

This section describes two suggested pipelines, depending on the two identified stages:
Stage 1: Set-up of the Visualisation and Analysis Environment
Stage 2: Automation of the Visualisation Pipeline

Both approaches make use of existing modes operations on supercomputing clusters - called *interactive* and *batch*.

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 *interactive* mode allows launching `ParaView` Graphical User Interface (GUI), where the user can tweak relevant settings live, before setting up any automated scripts.

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 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 best 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 3.1.2) will cover an alternative way to set-up the visualisation interactively, and then finally the next section [ref] will demonstrate how to combine the interactively generated state files or scripts with batch automation.

3.1.2 Method 2: Trace-file Generation

In addition to state files described in part one (Subsection [ref]), 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 [ref]. 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 (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 alone 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 applies the cross-section filter on 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*.

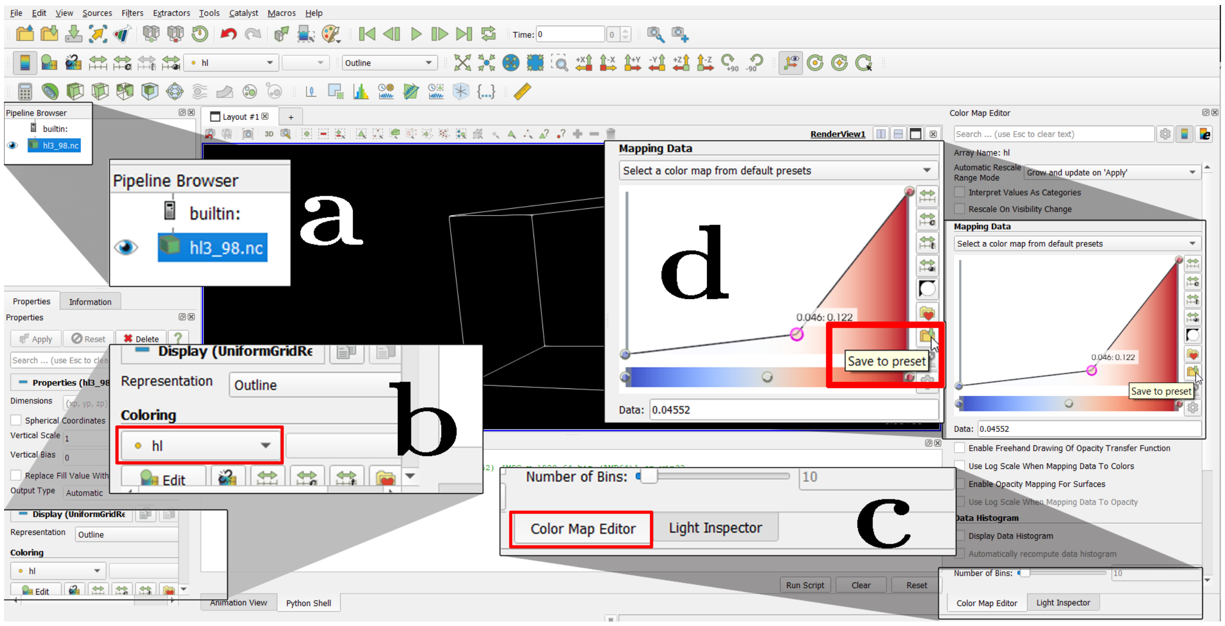

Figure 1: How to save custom transfer functions. Note, make sure there is a loaded dataset (a), the colouring field is set to a variable that has a valid range, such as 'hl' (b), then making sure that *Color Map Editor* Window is opened (c), and finally by clicking *Save as preset* button (d). The transfer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

After choosing one of the methods described in this Section [ref], we have either 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 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 (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), and using the GUI from the client (when server sends back rendered results as images to the client seamlessly, replacing the usual viewport).

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`)

4.1 `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.

4.2 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.

5 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` and `ParaView` batch programming did not work, therefore, after following the more general batch submission documentation on `ARCHER2` it was updated 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 here 0-15 is 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 [ref].

References

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