Report on the Catalyst implementation

In situ visualization of the iPIC3D simulation

Introduction

The aim of the internship project is the implementation of data visualization during numerical simulations. Until now, the only data available are the ones written on disk. As it is expensive in terms of storage and download, the simulation does not write every single time step. This results in less information and control. To both get the most out of every run and monitor the state of the simulation in between data writing, we implemented an *in situ* (during the simulation) data visualization using ParaView Catalyst.
ParaView Catalyst architecture was changed in Paraview 5.9 and, at the time this document is written, the version installed on TGCC Irene is Paraview 5.8. This is the reason why we decided to implement the Catalyst version prior to Paraview 5.9, called **Catalyst Legacy**. Note that since Paraview 5.9, a new version has been developed, called Catalyst V2, that is significantly different. We did not implement that version. However, the code is designed to be modular so that switching between the two should be easy, when Catalyst V2 is developed.

The structure of the Paraview Catalyst implementation, no matter the version used, is summarized in Figure 1. In addition to the code itself, there are 2 parts added to iPIC3D: the adaptor and a Python visualization script. The first one interacts with the code and prepares the data for the visualization, and the second one is responsible for setting up the rendering frequency and the view in general. Although the Python visualization script is generated automatically by Paraview locally, it nonetheless needs to be corrected/modified in some parts, as described in Section 1.6.

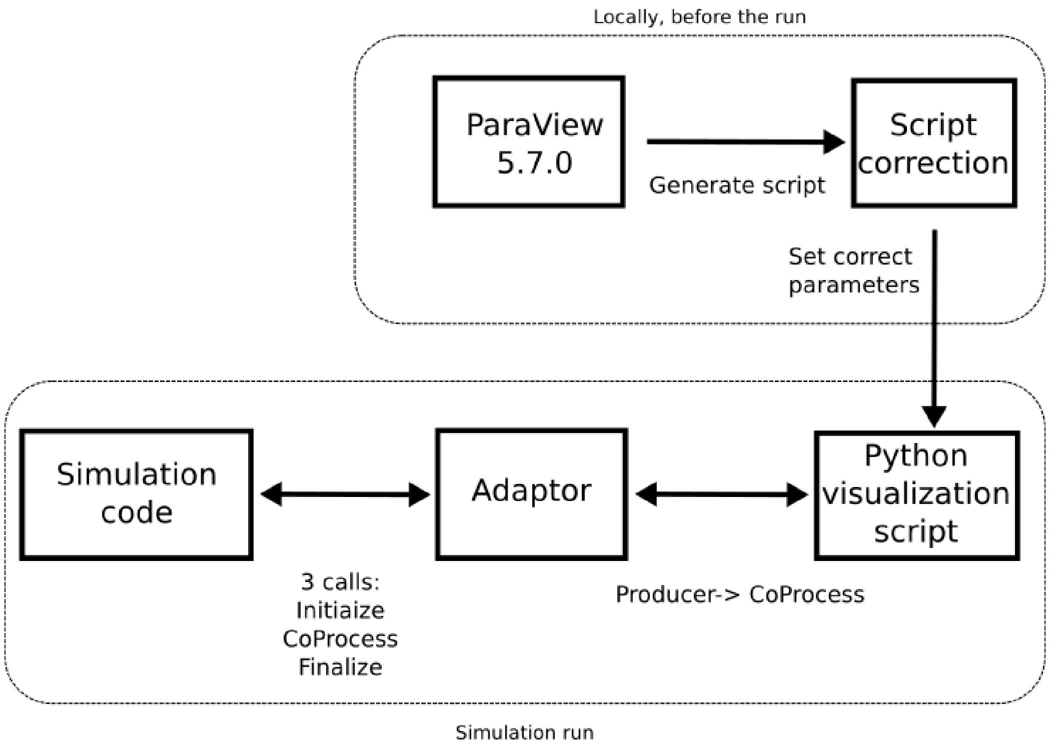


Figure 1.

To use Catalyst Legacy, there are some variables to set up, some changes in the code to understand and some tools to get used to. This report is meant to explain all of the above in a way that makes it easy using the new implementation. First, a User Guide explains how to set up a working implementation locally and how to set up the simulation on TGCC Irene. Second, a Development Document is provided which describes in more detail the routine and the changes in the iPIC3D code.

1 User Guide

1.1 Brief summary of successive steps to set up the simulation

The following is a brief description of how to set up the Catalyst Legacy implementation. Each step is described in a dedicated section.

1. Build Paraview 5.7.0 with Python, MPI and Catalyst enabled
2. In CMakeList.txt set the variable USE_CATALYST to "LEGACY" and the path to ParaView
3. On the local machine, open Paraview 5.7.0 and create the view
4. Generate and correct the Python script
5. Set the path to the Python script in the simulation input file
6. Run the simulation

1.2 Operating System

On a personal machine, the OS tested is **Ubuntu 18.04**. Few tests were made on Ubuntu 22.04 and were not successful: the code compiles but does not work because there are some critical issues in building the needed version of ParaView. If Ubuntu 18.04 is not available, the suggestion is to use a Virtual Machine (preferred option) or set up a docker image.

1.3 Building ParaView Catalyst Legacy

1.3.1 Personal machine

To run the simulation with *in situ* visualization enabled, we need to build one of these three Paraview versions: 5.7.0, 5.8.0, 5.8.1. The version recommended (tested) is **Paraview 5.7.0** and to build it we followed the instructions found here:
https://github.com/Kitware/ParaView/blob/release/Documentation/dev/build.md
Note that we are using an old ParaView version: not all the options exist or have the name reported on the github documentation. These are the steps to follow:

1. Install the dependencies:

   ```
   sudo apt-get install git cmake build-essential libgl1-mesa-dev libxt-dev
   qt5-default libqt5x11extras5-dev libqt5help5 qttools5-dev
   qtxmlpatterns5-dev-tools libqt5svg5-dev python3-dev python3-numpy
   libopenmpi-dev libtbb-dev ninja-build
   ```

   NOTE: In our case `libopenmpi-dev` generates errors when compiling the code. We use `mpich` instead and everything works fine. The `qt5` libraries are mandatory if you plan to use the graphical user interface (GUI) to generate the Python script. `Ninja` is optional, see below

2. Build 5.7.0 version with Ninja

```
git clone https://gitlab.kitware.com/paraview/paraview.git
mkdir paraview/build
cd paraview
git checkout v5.7.0
git submodule update --init --recursive
cd build
cmake      -GNinja      -DPARAVIEW_ENABLE_PYTHON=ON      -DPARAVIEW_USE_MPI=ON
-DPARAVIEW_ENABLE_CATALYST=ON ..
ninja
```

NOTE: The `-D` parameters are mandatory to set up a working version of Catalyst. Building with `Ninja` is not necessary, but is recommended as it speeds up the process.

1.3.2 Computing centre

Building a ParaView version in a supercomputer can be very tricky and different in every single case. For the OCA's mesocentre Licallo, the attempts are described in "Appendix A". For the TGCC Computing Center Irene, there is no need to install ParaView, instead it is recommended to load the suitable module.
The changes in the code to run the *in situ* visualization on Irene are described in the relative Section. The commands and modules to load are shown in the Section 1.9.

1.4 CMake file

After building Paraview Catalyst there are some parameters to set in the *CMakeList.txt* file, that deal with the version used and the path to the Paraview build.

1.4.1 Setting Catalyst version

The line that define the Catalyst version is:

```
set(USE_CATALYST "LEGACY")
```

LEGACY is the only working variable. Every other string will deactivate the *in situ* visualization.

1.4.2 Setting the path

After setting the Catalyst version, CMake needs to know where to find the `paraview-config.cmake` file. This is the reason why it is not possible to use a binary distribution of Paraview. `paraview-config.cmake` is stored in the ParaView build directory and the command to set the path to the directory is:

```
set(ParaView_DIR /home/../paraview/build )
```

In case of a pre build module (e.g. TGCC), the path is

```
set(ParaView_DIR
       /ccc/products/paraview-5.8.0/intel--19.0.5.281__openmpi--4.0.1/default
       /lib64/cmake/paraview-5.8)
```

We strongly recommend to set also CMAKE_CONFIG_PATH

```
list(APPEND CMAKE_MODULE_PATH
        "/ccc/products/paraview-5.8.0/intel--19.0.5.281__openmpi--4.0.1/defaul
        t/lib64/cmake/paraview-5.8")
```

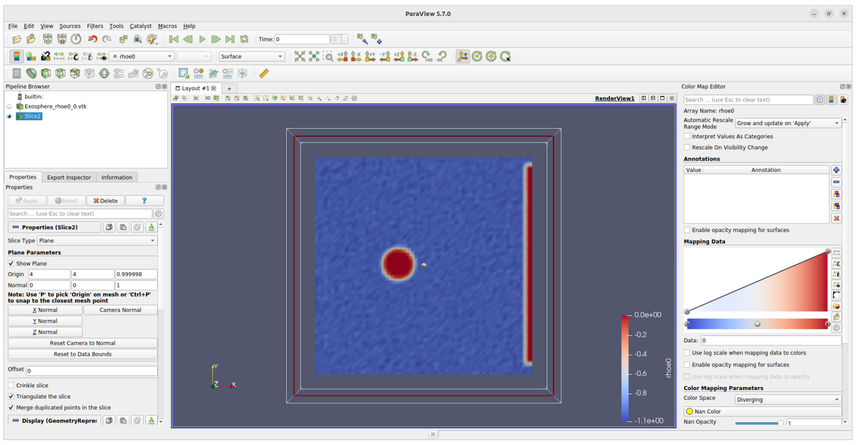
It is not necessary on a local machine, but it is recommended on supercomputers.

1.5 Generate a Catalyst Legacy script

We focus now on how to get the auto generated python script for Catalyst Legacy. The first thing to do is open Paraview 5.7.0. There is no need to use the build version, the binary code downloaded from https://www.paraview.org/download/ works. Any Paraview version that has the button "*Generate Script*" shall be fine. Version 5.4.0 does not have it, neither does 5.10, that is why 5.7.0 is recommended.

1.5.1 Set the view

In Paraview, we load the data and set the view we want to reproduce later within the simulation (Figure 2). To generate the script, it is not necessary to load the data of the future simulation. Instead, another dataset can be used to generate the script. This dataset must be the same kind of data (scalars, vectors, etc) but not necessarily the same grid size. In such case, some parameters need to be corrected within the generated visualization script.
For example, let us consider a Python script that sets up a slice in a certain position of interest, such as e.g. the planet position. If the data used for the generation of the visualization script are not representative of the data that will be generated by the simulation, so that the planet location is different, then the slice will not be in the desired position. The slice position then needs to be modified in the generated script. One could also update directly in the generated visualization script other features such as positions, color palette, etc.



Properties (Slice2)
Slice Type: Plane
Plane Parameters
✓ Show Plane
Origin: 4 | 4 | 0.999998
Normal: 0 | 0 | 1
Note: Use 'P' to pick 'Origin' on mesh or 'Ctrl+P' to snap to the closest mesh point
Offset: 0
☐ Crinkle slice
✓ Triangulate the slice
✓ Merge duplicated points in the slice

Array Name: rhoe0
Automatic Rescale Range Mode: Grow and update on 'Apply'
☐ Interpret Values As Categories
☐ Rescale On Visibility Change
☐ Enable opacity mapping for surfaces
Data: 0
☐ Use log scale when mapping data to colors
☐ Enable opacity mapping for surfaces
☐ Use log scale when mapping data to opacity
Color Mapping Parameters
Color Space: Diverging
Nan Opacity: 1

Figure 2.

1.5.2 Generate the Script from ParaView

Once the view is set, click on Catalyst and then on Generate Script (Figure 3).
This will open an initial window, press Next to continue.

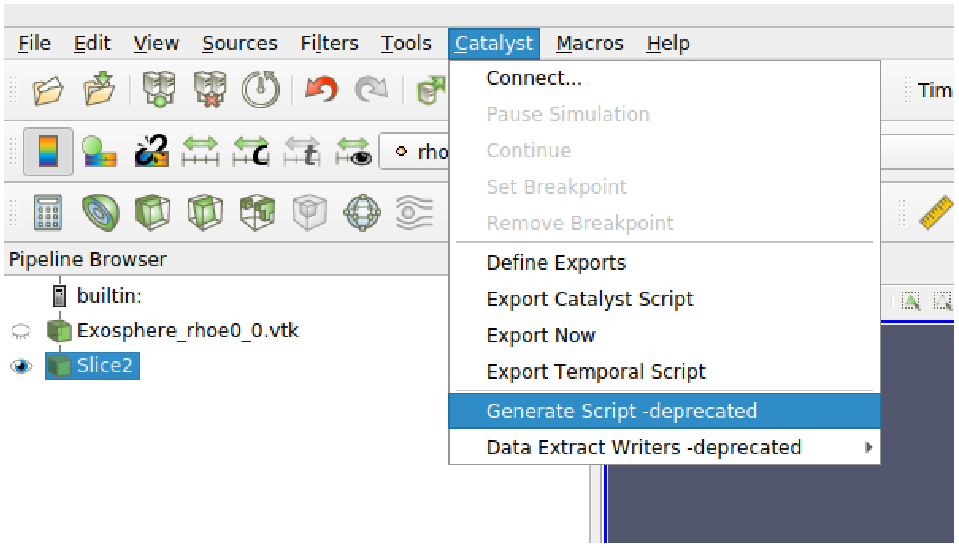
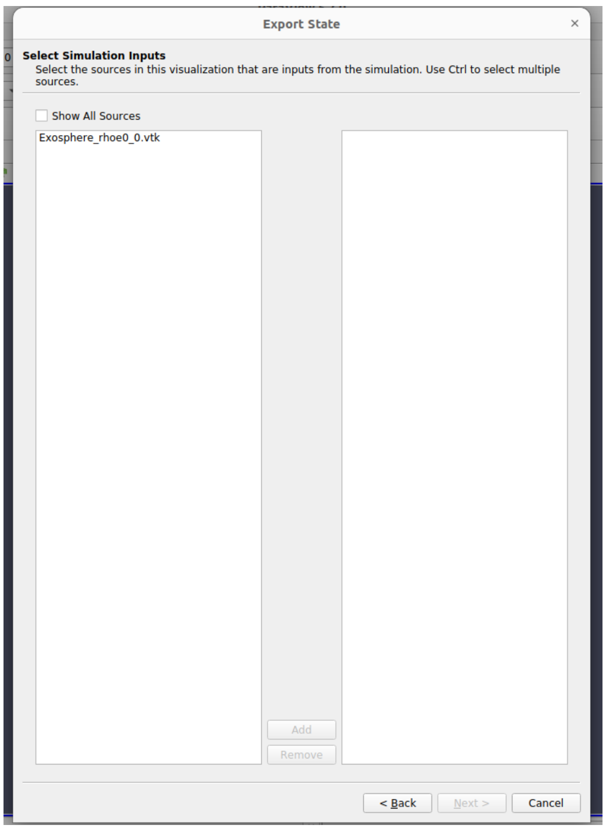


Figure 3.

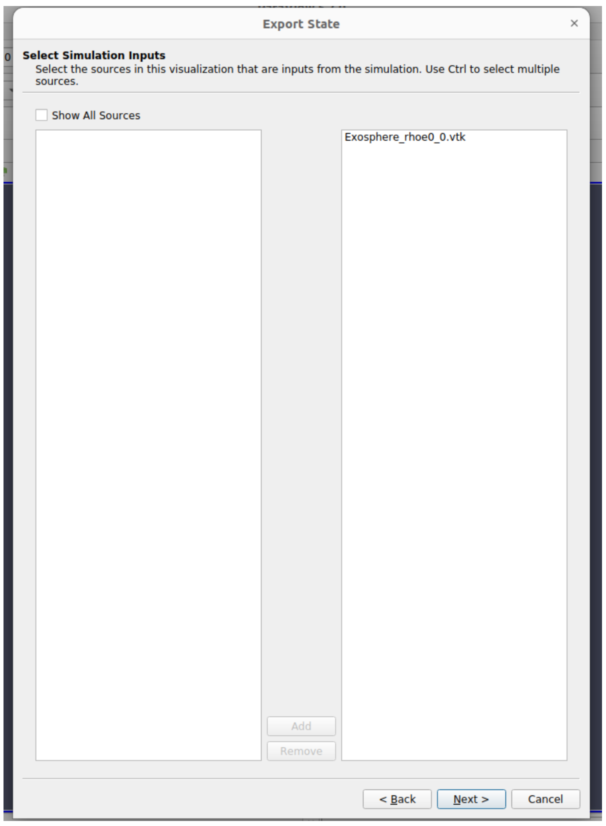
The user decides, on the new window, which data to get from the simulation. If the data is not selected, the generated script searches and loads the VTK file stored in memory. This could be practical for logos or images, but it has not been tested.


Export State

Select Simulation Inputs
Select the sources in this visualization that are inputs from the simulation. Use Ctrl to select multiple sources.

Show All Sources

Exosphere_rhoe0_0.vtk

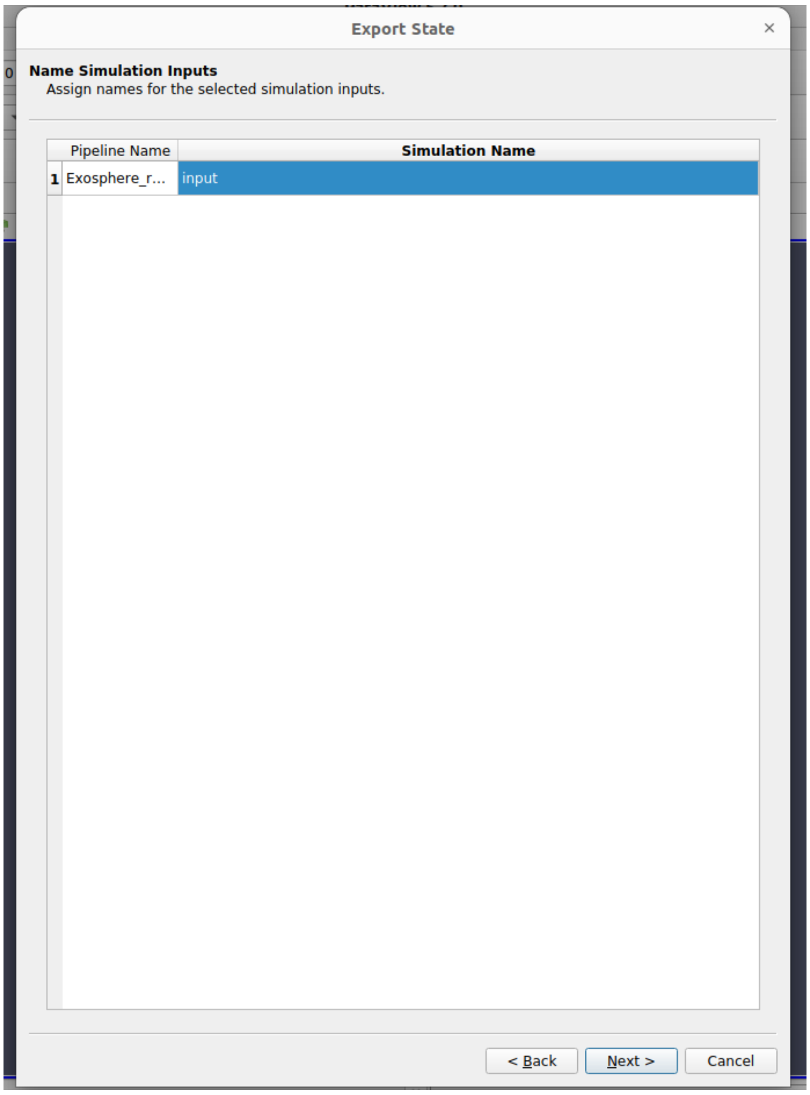

Export State

Select Simulation Inputs
Select the sources in this visualization that are inputs from the simulation. Use Ctrl to select multiple sources.

Show All Sources

Exosphere_rhoe0_0.vtk

After adding all the inputs needed, the user sets their name in the simulation code. On iPIC3D, because there is only one grid in the adaptor code and every single observable is inside this grid, the simulation name for every source must be named `input`. The name `input` is set by Inputname in adaptor_legacy.cpp.

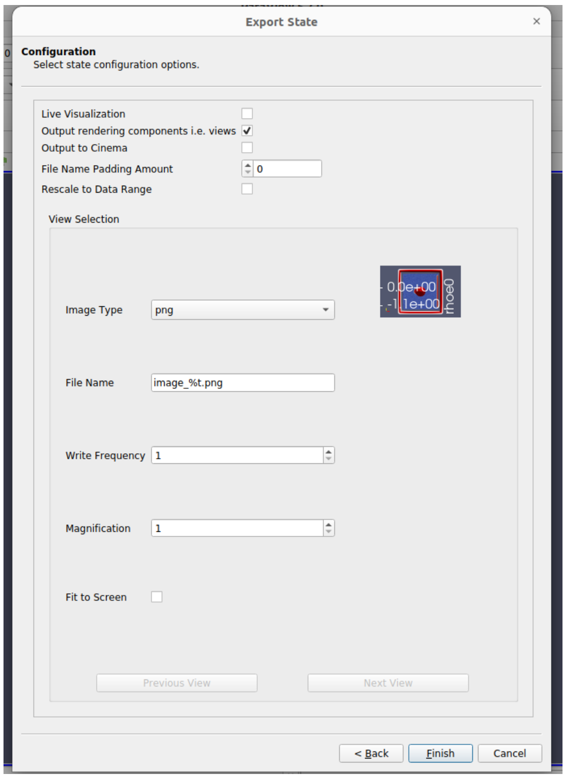


	Pipeline Name	Simulation Name
1	Exosphere_r...	input

Figure 4.

The last step is the configuration of the render (Figure 5). If none of the three options is selected, no render will be performed. The three options are:

- Live Visualization: enables you to see the data on Paraview on the local machine in "realtime". This is not possible in our case, due to connection restrictions. **It was not tested.**
- Output rendering components: render the set view. **This is the option to select!**
    - *padding*: used for the name of each generated image (ex: *padding*=3 means that the image generated during cycle=45 is named image045). Note that *padding* must be set larger or equal to log10(total number of cycles)+1
    - *frequency*: the image is generated every *frequency* cycles (ex: *frequency=10* means that the image will be generated every 10 simulation time steps)
- Output to cinema: should move the camera position and filters during the simulation. **It was not tested.**



Configuration
Select state configuration options.

Live Visualization ☐
Output rendering components i.e. views ✓
Output to Cinema ☐
File Name Padding Amount 0
Rescale to Data Range ☐

View Selection

Image Type png
File Name image_%t.png
Write Frequency 1
Magnification 1
Fit to Screen ☐

Figure 5.

After these steps the script is generated and saved in the folder selected by the user.

1.6 Correcting the Python script

The auto generated Python script generally works out of the box if the code is "easy". Unfortunately, there is no way to know in advance if the script is fine or if it will generate an error during the run. It is therefore advised to systematically test and validate the python script on a fast simulation before starting a production run. In Appendix B, there is a comment on the different parts of the script that may help in case of an error. We have found no helpful reports and imagination is a key ingredient!

1.6.1 Root Directory

To define where to store the products (images, plots,....) set the `rootDirectory` :

```
# a root directory under which all Catalyst output goes
rootDirectory='./data/images'
```

1.6.2 Image dimension

Ffmpeg and other video generators will not work if one of the two dimensions of the image is odd (ex: an image of 480x480 pixels works, but not 480x481 pixels). To set the resolution to HD (1280x720) there are 2 points to the script. In "Render View":

```
# Create a new 'Render View'
     renderView1 = CreateView('RenderView')
     renderView1.ViewSize = [1280, 720]
```

and in “Register View”:

```
coprocessor.RegisterView(renderView1,filename='image_%t.png',freq=1,
                    fittoscreen=0, magnification=1, width=1280, height=720,
                    cinema={}, compression=-1)
```

1.6.3 Output frequency

The frequency at which the data are rendered (called *frequency* by ParaView when generating the script) is now called `freq` in “RegisterView” (see above) and in:

```
# these are the frequencies at which the coprocessor updates.
 freqs = {'input': [1, 1, 1]}
```

In this example, the image is saved at each simulation cycle.
There should be as many numbers as the number of sources rendered (rho, B, slices, stream tracer and so on). To change the *frequency*, without generating again the script, set all the values in RegisterView and in “freqs” accordingly. It is strongly advised to always use the same frequency for all sources.

1.6.4 Create Producers

The Producer is responsible for the rendering. It is initialized by:

```
# create a producer from a simulation input
exosphere_Je_ = coprocessor.CreateProducer(datadescription, 'input')
```

Creating a Producer is the same action as clicking on “Apply” in the ParaView GUI when opening a new file. For this reason, there is a `CreateProducer` for every source selected in the view. Because iPIC3D has a single grid, there is no need for multiple Producers and the user can use a single Producer. For example:

```
# create a producer from a simulation input
exosphere_Je_ = coprocessor.CreateProducer(datadescription, 'input')

# create a new 'Stream Tracer'
streamTracer2 = StreamTracer(Input=Data,
                          SeedType='High Resolution Line Source')

# create a producer from a simulation input
legacyVTKReader1 = coprocessor.CreateProducer(datadescription, 'input')

# create a new 'Slice'
slice1 = Slice(Input=legacyVTKReader1)
```

creates two Producers with the same grid, it is advised to delete one of the two Producers and rename the remaining Producer “Data”:

```
# create a producer from a simulation input
Data = coprocessor.CreateProducer(datadescription, 'input')

# create a new 'Stream Tracer'
streamTracer2 = StreamTracer(Input=Data,
                          SeedType='High Resolution Line Source')
```

```
# create a new 'Slice'
slice1 = Slice(Input=Data)
```

Using one producer is not mandatory, sometimes it works with more than one, sometimes it does not. As a good habit, always use just one Producer.

1.6.5 Parameters

Filters and colors transfer functions have parameters that depend on the simulation cube and/or input file. Remember to change the parameters (ex. positions, color bars, etc) in case the data uploaded on the ParaView GUI are not the same as the one expected from the simulation!

1.7 Input file

The path to the python script is set in the input file:

```
ParaviewScriptPath  = /home/.../script.py
```

In the script directory there are some python scripts that can be used to test the implementation. The list of the generated script is in Table 1. Note that they have specific values for the position of the slice and the color bar interval. Check that the values agree with the input file of the simulation.

Python script name	freq	type	position (x,y,z)	note
no_render.py	-	-	-	no image
no_data.py	1	-	-	just background
B.py	1	Slice, plane xy	(4,4,1)	show cube and slice
rhoe0.py	1	Slice, plane xz	(4,4,1)	
rhoi1.py	1	Slice, plane xy	(4,4,1)	show cube and slice
Tcart_e0.py	1	Slice, plane xy	(4,4,1)	component XX
Tcart_e0_YY..py	1	Slice, plane xy	(4,4,1)	component YY
Tperpar_e0.py	1	Slice	centre	no actual slice
Tperpar_e0_YY.py	1	Slice, plane xy	(4,4,1)	
Ve0.py	1	stream tracer	(1,4,1)	

Table 1.

NOTE: the component to show is selected in the color transfer function using `_eLUT.VectorComponent = ...` . If `VectorComponent` is not set, the first component, that is 0, is used by default.

1.8 Run the simulation on the local machine

Once all the steps above have been completed, you can compile the code and run the simulation with mpiexec or mpirun:

```
mpiexec -n 8 ./iPIC3D ../inputfiles/testMagnetosphere3D_small.inp
```

Note: hdf5 raises an error if there is no directory called “data”.

1.9 TGCC Irene

To run the *in situ* visualization on TGCC Computing Center Irene these are the steps to follow:

1. Load the modules
2. Request a dedicated node on the graphical machine
3. Connect to the node via Licallo
4. Perform the simulation

1.9.1 Load the modules

The modules to load are strictly related to the paraview module.

```
#iPIC3D
module load intel/19.0.5.281
module load mpi/openmpi/4.0.2
module load flavor/hdf5/parallel
module load hdf5/1.8.20

#paraview
module load llvm cmake/3.13.3 ospray embree boost paraview/5.8.0
```

NOTE: All the modules are strictly necessary. The user can change `cmake` to use a newer version, not an older one.

As described in “Setting the path”, the path to the `paraview-config.cmake` file is:

```
/ccc/products/paraview-5.8.0/intel--19.0.5.281__openmpi--4.0.1/default
/lib64/cmake/paraview-5.8
```

Note: In the flavor modules there is `flavor/paraview/osmesa`. We tried to use that flavor, setting the relative path, but it did not work. TGCC Hotline reported an error during the compilation of the flavor and does not allow anyone to use it.

1.9.2 v100l

v100l is a partition of Irene with mixed CPUs and GPUs. The hardware configuration is reported in Table 2:

Partition name	v100l
CPUs	2 x 18 cores Intel Cascadelake@2.6GHz x 2 hyperthreads
GPUs	1x Nvidia Tesla V100-PCIE
RAM/Node	360 GB
RAM/Core	10 GB

Table 2.

To request a dedicated node on the partition the command is:

```
ccc_visu virtual -p v100l -A gen12428
```

NOTE: gen12428 is the name of the project allocation on the supercomputer. To be changed in case of a new allocation of resources in the future.

Hit CTL-D or enter "exit 0" to close the interactive and visualization sessions and release the allocation.

At the time this document is written the public documentation of Irene does not include ccc_visu. It should be included in the september/october 2022 version.

1.9.3 Connect to v100l

The `ccc_visu` command generates a link such as:

```
https://visu-eu.ccc.cea.fr/visu-eu/...
```

To connect to the visualization session open another terminal and connect to Licallo using -X option. On Licallo open firefox and go to the link generated by `ccc_visu`. Enter the Irene credentials then click on the button to open a visualization session on the browser. The browser page will show the GUI of the node reserved on v100l.
In the GUI, open a terminal window and run the simulation in the same way it is done on the local machine.

Unfortunately, we are not able to show images of the GUI of Irene v100l: there are some technical issues (`"An internal server error has occurred"`).

1.9.4 Running the simulation

Before opening a terminal window, in the GUI, to run the simulation, there are some lines to comment or change in Adaptor_legacy.cpp. We can not explain why these lines raise an error, we just found that without them the code works, though it renders just the background.

In CoProcess, comment all the lines that insert a value and add an array to the grid. For example:

```
//  fd1->InsertNextValue(timeStep);
//  VTKGrid->GetFieldData()->AddArray(fd1);
```

In UpdateVTKAttributes, disable the cycle that sets the values of the arrays point by point. To do that, use 0 instead of ns in the *for* cycle of the species:

```
for(int si = 0; si < 0; si++){
```

With this change there is no way to render any data: all the array values are zero. Unfortunately this is the only way we found to render an image, proving that all the libraries needed are installed.
On v100l the command syntax to run the simulation is exactly the same as the command syntax on the local machine.

1.9.5 Problems with this implementation

There are two different problems with this implementation. The first problem is that the only image that can be rendered is the background image without any data (python visualization script *no_data.py*). An example is in Figure 6. All the visualization scripts with data from the simulation raise an error and do not render any image.

Figure 6.

We did not find a solution to this problem because the second one made using this implementation pointless, even if it was working. v100l and all the other mixed partitions of Irene are not suitable for a production run: there are not enough cores! Nonetheless, we were able to run a small weak scaling test and the results are described in section 2.4.

2 Development Report

2.1 Changes in CMakeList.txt

In order to communicate to the compilers the status of the Catalyst implementation, we added this *if structure* in the CMake file:

```
if(USE_CATALYST STREQUAL "LEGACY")
   add_definitions(-DUSE_CATALYST_LEGACY) #define USE_CATALYST_LEGACY
endif()
```

The command `add_definition` defines the variable after the -D in the code. The defined variable is passed to the preprocessor when compiling the code. In this way

```
#ifdef USE_CATALYST_LEGACY
```

is set to true in the code. All parts dedicated to the *in situ* visualization are within an `#ifdef` statement that is set to true when `USE_CATALYST` is set to `LEGACY`. The developer can therefore easily turn off and on Catalyst by changing `USE_CATALYST` in CMakeList.txt. The same idea is used inside CMakeList.txt itself for the parts needed just by the *in situ* visualization (e.g. linking libraries). The same structure can be used for the implementation of version V2.

The default Python version is Python3. To change it the line is:

```
target_link_libraries(iPICAdaptor PRIVATE ParaView::PythonCatalyst
                                  VTK::CommonDataModel Python3::Python)
```

Instead of `Python3::Python` use `Python::Python` or `Python2::Python`, depending on the Python name in the system.

2.2 Catalyst in the simulation code

The general idea of Paraview Catalyst is to change the simulation code the least possible. We only added four calls to the Adaptor functions in `iPIC3dlib.cpp.`

2.2.1 Modularity

To be able, in the future, to have a code with both versions, every call to Adaptor Legacy functions is inside a `ifdef` statement. The future implementation of V2 must continue using `ifdef` statements for each call, in order to use the modularity of the architecture. With this approach, the variable `USE_CATALYST` controls if Catalyst is enabled and which version to use: legacy or V2.

2.2.2 Calls to Adaptor functions in iPIC3Dlib.cpp

The first step is adding the header :

```
#ifdef USE_CATALYST_LEGACY
   #include "Adaptor_legacy.h"
#endif
```

In `int c_Solver::Init`, there is the call to initialize the Processor, that will handle the data, and the VTK grid, that will store them.

```
#ifdef USE_CATALYST_LEGACY
Adaptor_legacy::Initialize(col, \
    (int)(grid->getXstart()/grid->getDX()), \
    (int)(grid->getYstart()/grid->getDY()), \
    (int)(grid->getZstart()/grid->getDZ()), \
    grid->getNXN(),
    grid->getNYN(),
    grid->getNZN(),
    grid->getDX(),
    grid->getDY(),
    grid->getDZ());
#endif
```

This call is for the 3D physical space, but can be modified to the velocity or the phase space. The complete explanation on this function parameter is in the Subsection 2.3.1.

In `c_Solver::~c_Solver()` we finalize the Processor and grid.

```
#ifdef USE_CATALYST_LEGACY
Adaptor_legacy::Finalize();
#endif
```

The post-processing and rendering of the data is inside `void c_Solver::WriteOutput`.

```
#ifdef USE_CATALYST_LEGACY
Adaptor_legacy::CoProcess(col->getDt()*cycle, cycle, EMf);
#endif
```

2.3 The Adaptor Legacy code

We now explain the four functions contained inside Adaptor_legacy.cpp.

2.3.1 Initialize

This is the function that sets the Processor, the VTK grid and the pipeline. Its parameters are:

```
void Initialize(const Collective *sim_params, const int start_x,
               const int start_y, const int start_z, const int nx,
               const int ny, const int nz, const double dx, const double dy,
               const double dz)
```

We use two lines to initialize the processor

```
Processor = vtkCPProcessor::New();
Processor->Initialize();
```

After that, we need to initialize and add the pipeline to the processor, as follows:

```
vtkNew<vtkCPPythonScriptPipeline> pipeline;
```

```
pipeline->Initialize(sim_params->getParaviewScriptPath().c_str());
Processor->AddPipeline(pipeline);
```

NOTE: `getParaviewScriptPath().c_str()` returns the script path set in the input file. The function `getParaviewScriptPath` is defined in Collective.h.

The parameters to initialize the VTK grid sets the 3D physical space:

```
VTKGrid = vtkImageData::New();
VTKGrid ->SetExtent(start_x, start_x + nx - 3, start_y , start_y + ny - 3,
                      start_z, start_z + nz - 3);
VTKGrid ->SetSpacing(dx, dy, dz);
```

NOTE: “- 3” is due to the ghost cells in each MPI process of iPIC3D. It is possible to set another space, that is not the 3D physical one (e.g. velocity space, phase space), by setting different parameters when calling the function.

2.3.2 CoProcess

This function is called within iPIC3D at each timestep, no matter the set visualization output *frequency*. It is responsible for setting the parameters of the cycle and rendering the data.

```
void CoProcess(double time, unsigned int timeStep, EMfields3D *EMf)
```

Apart from the various parameters, the rendering is done by:

```
if (Processor->RequestDataDescription(dataDescription) != 0) {
   vtkCPInputDataDescription *idd =
       dataDescription->GetInputDescriptionByName(InputName);
   UpdateVTKDataStructures(idd, EMf);
   idd->SetGrid(VTKGrid);
   Processor->CoProcess(dataDescription);
 }
```

NOTE: `InputName` has to be the one used in the Python script. `InputName is` set to `input`.

2.3.3 UpdateVTKAttributes

This is the function that updates the dataset values. As a first step, all the needed arrays are initialized. Hereafter, we use B as an example:

```
vtkNew<vtkDoubleArray> field_array_B;
field_array_B->SetName("B");
field_array_B->SetNumberOfComponents(3);
field_array_B->SetNumberOfTuples(VTKGrid->GetNumberOfPoints());
vtk_point_data->AddArray(field_array_B);
```

In the case of temperature or density, for which one array for each species is required, the implementation cycles over the species, changing the name of the array.

In order to avoid initializing arrays each cycle, after the first cycle, the arrays are downcasted:

```
        vtkDoubleArray *field_array_B =
                 vtkDoubleArray::SafeDownCast(vtk_point_data->GetArray("B"));
```

and then we set the values of each component in each point :

```
    for (vtkIdType p = 0; p < VTKGrid->GetNumberOfPoints();' ++p) {
           // Get cells's indices i, j , k
           const size_t k = p / (dims[0] * dims[1]);
           const size_t j = (p - k * dims[0] * dims[1]) / dims[0];
           const size_t i = p - k * dims[0] * dims[1] - j * dims[0];


           field_array_B->SetComponent(p, 0, Bx[i+1][j+1][k+1]);
           field_array_B->SetComponent(p, 1, By[i+1][j+1][k+1]);
           field_array_B->SetComponent(p, 2, Bz[i+1][j+1][k+1]);
```

NOTE: +1 manages the ghost cell and is related to - 3 in the Initialize function.

2.3.4 Notes on the Adaptor code

The implementation was designed with just one grid to avoid redundant grids. If more than one grid is needed, the developer is invited to refer to the example CxxMultiChannelInputExample inside the paraview Example/Catalyst directory. That directory contains useful examples that can be used to study different implementations.
The command `SetComponent` copies the data in a new variable, value by value. A faster way exists (although it is not working in the current implementation) that you can find in the comments that uses `SetArray`. To use `SetArray` the memory layout of the variable has to be known in advance. There is no need for the faster version because we render the fields and not the particles. In Catalyst V2 it seems that the data is not copied but rather it uses pointers, therefore should be faster.

2.4 Weak Scaling on v100l

As shown in the section "TGCC Irene", we were able to render images, however with no data included, using v100l.
To check the time consumption of the render, we performed a weak scaling using 2 to 32 processes, using a fixed grid size and particle per cell number in each process, increasing the size of the box accordingly. We simulated 50 cycles and rendered one image each cycle. In Table 3 are summarized the parameters used.

cores	cycles	render frequency	nxc	nyc	nzc	XLEN	YLEN	ZLEN	time render
2	50	1	32	16	16	2	1	1	7,11207
4	50	1	32	32	16	2	2	1	9,95697
8	50	1	32	32	32	2	2	2	12,7738
16	50	1	64	32	32	4	2	2	14,6022
32	50	1	64	64	32	4	4	2	16,336

Table 3.

The comparison between the simulation run with and without render is shown in Figure 7, using the time values in Table 4. Note that the initialization and the finalization of the runs are both included in the computational time. Note also that the scaling stops at 32 processes because there are just 36 processes in the v100l node.

cores	time no_render	time render
2	7,10554	7,11207
4	8,09942	9,95697
8	9,60479	12,7738
16	11,8864	14,6022
32	15,5884	16,336

Table 4.

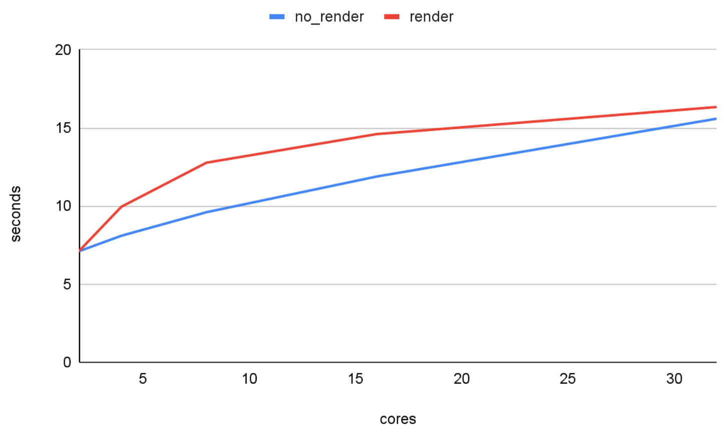


Figure 7.

The relative time consumption of the render is computed using the equation below: $time\ consumption = (t_{render} - t_{norender})/t_{norender}$

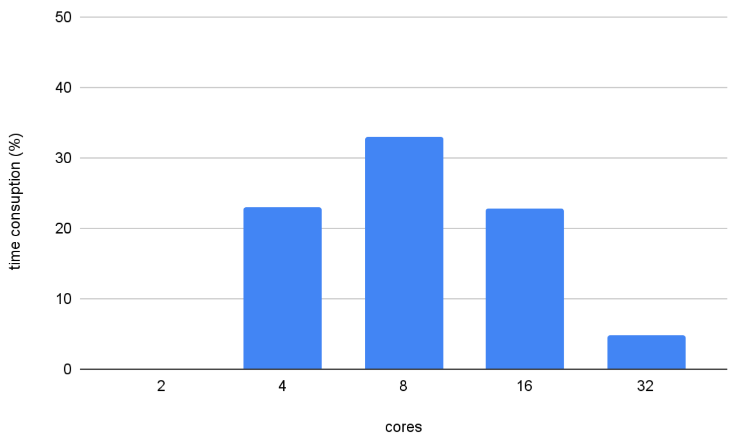


Figure 8.

The relative computational time used for *in situ* visualization is below 5% of the total CPU time for 32 processes. We expect the same performance for a larger number of processes, though this remains to be validated. Note that we tested the rendering of one background image per cycle. The relative computational time used for in situ visualization with postprocessing and rendering of the datacube remains to be validated. Time consumption of multiple images needs to be tested as well. See also the next Section.

2.5 Future development

2.5.1 Use only CPUs

The procedure described in Section 1.9 is not suitable for a production run. To extract all the benefits from the *in situ* visualization, the implementation must use a partition with an adequate amount of CPUs (greater than 10^4). There are no partitions in Irene that have such a number of CPUs linked to GPUs for the rendering and visualization part. The only solution is to avoid the ParaView module and build it as an external library. Our attempt in building ParaView is described in Appendix C: TGCC Irene.

2.5.2 Scaling tests

As described at the end of Section 2.4, there are tests that have to be performed. The *in situ* visualization has to be tested with more than 32 processes, with data rendering (i.e. not just rendering of the background) and with multiple images rendered at the same time. To run these tests Subsection 2.5.1 must be implemented and the error generated while rendering data must be corrected.

2.5.3 Dynamic change of the visualization scripts

To simplify the generation and use of the script, we suggest developing a c++ code that dynamically changes the slice positions and the range of the color transfer functions in the visualization script. This code would automate the procedure explained in Section 1.6.

2.5.4 Implementation of Catalyst V2

It is very plausible that the ParaView modules will be updated to a newer version. When that happens, it will be possible to use the new version: Catalyst V2. The implementation and steps described in this document do not work with the new version since it is based on conduit nodes, instead of VTK libraries, to pass the information between the c++ adaptor and the Python visualization script. A good point to start is: https://discourse.paraview.org/t/how-to-try-out-catalyst2/9008. To learn more about V2:
https://gitlab.kitware.com/paraview/paraview/-/blob/release/Utilities/Doxygen/pages/CatalystPythonScriptV2.md

2.5.5 Personal comment for future work

The implementation would not have been possible without the adaptor example in the KTH iPIC3D code. We kept some of the comments that we think can be useful for future development. Being able to use KTH's code has been very helpful because the examples and documentation on Catalyst

Legacy are old and incomplete. The support page specific for *in situ* visualization is: https://discourse.paraview.org/c/in-situ-support/8/l/latest. Unfortunately it is not enough.
ParaView company, Kitware, makes money out of prepaid support and training. It is our personal opinion that this is the reason why the documentation is incomplete.

When planning future development of *in situ* visualization using ParaView Catalyst, we recommend considering paying for some kind of support. Otherwise there is the possibility of errors being generated without clear reasons or easy fix. These errors usually result in many hours spent debugging, especially when installing the first time catalyst on a new machine.

Appendix A: Licallo

We were not able to have a working code on Licallo and this appendix is meant to show the attempts made and the errors that we faced.
We followed the steps above to build ParaView v5.7.0. Since we don't need the GUI, we removed qt5 to avoid building also qt5:

```
cmake -GNinja -DPARAVIEW_ENABLE_PYTHON=ON -DPARAVIEW_USE_MPI=ON
       -DPARAVIEW_ENABLE_CATALYST=ON -DPARAVIEW_BUILD_QT_GUI=OFF ..
```

The compilation was long but at last it worked. After setting the right path and adding the needed libraries to the path, the code compilation (*cmake* and then *make*) worked. When running the code, the error was:

vtkXOpenGLRenderWindow (0x1923a290): Unable to find a valid OpenGL 3.2 or later implementation. Please update your video card driver to the latest version. If you are using Mesa please make sure you have version 11.2 or later and make sure your driver in Mesa supports OpenGL 3.2 such as llvmpipe or openswr. If you are on windows and using Microsoft remote desktop note that it only supports OpenGL 3.2 with nvidia quadro cards. You can use other remoting software such as nomachine to avoid this issue.

The OpenGL library is provided by the manufacturer of the graphics hardware. Since there is no graphical interface in a CPU computational cluster, there is no such library. We needed to install a specific version of a graphics library (Mesa) developed for machines with no graphics hardware: OSMesa. The idea was to build ParaView for "headless machine" (no screen) using OSMesa for the render, but we were not able to build OSMesa.
We followed the steps explained here: https://www.paraview.org/Wiki/ParaView_And_Mesa_3D
At first, we installed LLVM libraries. Due to older versions of CMake, Python etc, we were forced to use an older version and the installLLVM.sh used is:

```
#!/bin/bash

#Download the LLVM source
curl -L -O
       https://github.com/llvm/llvm-project/releases/download/llvmorg-8.0.1/l
       lvm-8.0.1.src.tar.xz

# Extract
tar -xvf llvm-8.0.1.src.tar.xz

# Configure
mkdir llvm
mkdir llvm-8.0.1.build
cd llvm-8.0.1.build
cmake                                                  \
  -DCMAKE_BUILD_TYPE=Release                           \
  -DCMAKE_INSTALL_PREFIX=/data/home/lquerci/Sources/llvm \
  -DLLVM_BUILD_LLVM_DYLIB=ON                           \
  -DLLVM_ENABLE_RTTI=ON                                \
  -DLLVM_INSTALL_UTILS=ON                              \
  -DLLVM_TARGETS_TO_BUILD=X86                          \
  -DLLVM_TEMPORARILY_ALLOW_OLD_TOOLCHAIN=ON            \
  ../llvm-8.0.1.src

# Build and install
make -j8
make install
```

After LLVM, we tried to install Mesa. However, it requires meson to be installed, that is not available on Licallo. We installed an old version of meson due to Python version problems and added the directory to the Python path. installMeson.sh :

```
#!/bin/bash

#Download and extract the latest Mesa source
curl -L -O
      https://github.com/mesonbuild/meson/releases/download/0.61.5/meson-0.6
      1.5.tar.gz
tar -xvf meson-0.61.5.tar.gz
```

For the same reasons, we installed an older version of Mesa. installMesa.sh:

```
#!/bin/bash
#Download and extract the latest Mesa source
#curl -L -O https://archive.mesa3d.org/mesa-18.0.0-rc5.tar.xz
#tar -xvf mesa-18.0.0-rc5.tar.xz

# Configure Mesa
#mkdir mesa
#mkdir mesa-18.0.0-rc5-build
cd mesa-18.0.0-rc5-build

meson.py                        \
  ../mesa-18.0.0-rc5            \
  --buildtype=release           \
  --prefix=/data/home/lquerci/Sources/mesa    \
  -Dvulkan-drivers=             \
  -Ddri-drivers=                \
  -Dgallium-vdpau=false         \
  -Dgallium-xvmc=false          \
  -Dgallium-omx=false           \
  -Dgallium-va=false            \
  -Dgallium-xa=false            \
  -Dgallium-nine=false          \
  -Dgallium-opencl=disabled     \
  -Dbuild-tests=false           \
  -Degl=false                   \
  -Dgbm=false                   \
  -Dglx=gallium-xlib            \
  -Dplatforms=x11               \
  -Dglvnd=false                 \
  -Dosmesa=gallium              \
  -Dopengl=true                 \
  -Dgles1=false                 \
  -Dgles2=false                 \
  -Dshared-glapi=true           \
  -Dllvm=true                   \
  -Dgallium-drivers=swrast,swr
 # -Dshared-llvm=true            \

ninja
ninja install
```

Because we used an old version of Mesa, some parameters have been changed. Unfortunately, this generates a cascade of errors that we tried to overcome. We installed mako via pip:

```
pip install -targhet:/data/home/lqueri/Source/ mako
```

and added it to the Python path. Then, we commented two if statements in meson.build, but it was not enough. We stopped when it was not possible to trace the error inside the nested build files.
Due to the fact that there is little support for this kind of implementation on the mesocentre, we decided to focus on Irene.
For clarity, this is part of the .bashrc file:

```
export MPICH_ROOT=$HOME/Sources/mpich/3.2
export
PATH=${PATH}:$HOME/Sources/mpich/3.2:$HOME/Sources/mpich/3.2/bin:$HOME
/Sources/mpich/3.2/include
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$HOME/Sources/mpich/3.2/lib
export MPICH_ROOT_DIR=$HOME/Sources/mpich/3.2
export MPICH_HOME=$HOME/Sources/mpich/3.2

export HDF5_ROOT=$HOME/Sources/hdf5/1.8.13
export
PATH=${PATH}:$HOME/Sources/hdf5/1.8.13:$HOME/Sources/hdf5/1.8.13/bin:$
HOME/Sources/hdf5/1.8.13/include
export
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$HOME/Sources/hdf5/1.8.13/lib
export HDF5_ROOT_DIR=$HOME/Sources/hdf5/1.8.13
export HDF5_HOME=$HOME/Sources/hdf5/1.8.13

export PATH=${PATH}:$HOME/Sources/meson-0.61.5

export LLVM_ROOT=$HOME/Sources/llvm
export
PATH=${PATH}:$HOME/Sources/llvm:$HOME/Sources/llvm/bin:$HOME/Sources/l
lvm/include
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$HOME/Sources/llvm/lib
export LLVM_ROOT_DIR=$HOME/Sources/hdf5/1.8.13
export LLVM_HOME=$HOME/Sources/llvm

export PYTHONPATH=${PYTHONPATH}:$HOME/Sources/
export PYTHONPATH=${PYTHONPATH}:$HOME/Sources/meson-0.61.5/
```

Appendix B: Comments on the Python script

There is little documentation on the Python visualization script. The list of community support for ParaView users is: https://www.paraview.org/community-support/. Unfortunately, it is very generic and it is hard to find the information needed. In order to understand the little information available, we describe briefly here the auto generated python script.
The auto generated script has the following structure:

- Heading: define some constants and the Root Directory
- CreateCoprocess: defines pipeline and sets the frequency of the updates and more
- Two more methods: RequestDataDescription and DoCoProcessing

We will focus on the definition of the Pipeline Class that is inside CreateCoprocess and is the one responsible for setting the view. The following is its structure.

```
# ----------------------------------------------------------------
# setup views used in the visualization
# ----------------------------------------------------------------
```

It defines the views used. It sets the camera position, axes, background and so on. The registration of the view is in this part and it is responsible for setting the resolution of the images and their frequency.

```
# ----------------------------------------------------------------
# setup the data processing pipelines
# ----------------------------------------------------------------
```

It defines all the post processes (slice, stream tracer, etc…). In order to show the right physical quantities, the name in the script and the name in the adaptor must match. If they do not, no error will be generated and the data will be zeroes. This applies also to the render part

```
# ----------------------------------------------------------------
# setup the visualization in view 'renderView1'
# ----------------------------------------------------------------
```

It defines the representation, the color transfer functions and, in general, the appearance of every object of the pipeline and the render. Remember that an object is rendered if it is set to Show.

Known problems are related to the opacity of rendered objects. For example, rendering a sphere with opacity level 0.8 will generate problems in the next objects to render, but the sphere will be rendered. These kinds of problems are hard to manage, for this reason we strongly encourage the user to always test the pipeline on the local machine!

Appendix C: TGCC Irene

In Subsection 1.9.4, we explain why the implementation using the paraview module is not suitable for the production run. As anticipated in Subsection 2.5.1, this appendix will describe our attempt to build ParaView from the git repository. We tried to follow the steps explained in Subsection 1.3.1. Unfortunately the *git clone* command did not work and so we cloned the ParaView source code on Licallo and then, using *scp*, we copied it on Irene. The Paraview 5.7.0 build directory occupies more than 7 GB of memory. Since the HOME file system maximum capacity (called *quota*) is 5 GB, we used the WORK file system instead.
To build ParaView we used Ninja. There are some problems with the ninja module. Therefore we strongly recommend trying to use *make*, it should resolve some of the problems we faced. The modules loaded:

```
#building paraview from git repository
module load intel/20.0.0
module load mpi/openmpi/4.0.2
module load flavor/hdf5/parallel
module load hdf5/1.8.20
#module load cmake/3.13.3 llvm qt python3
#module load ninja
```

`Ninja` needs at least `intel/20.0.0` but it is in conflict with the compiler needed by mpi (check *module show mpi/openmpi/4.0.2* to learn more). Building ParaView with a set of compilers (20.0.0) and then compiling the code with another set (19.0.5.281) is a critical point and a source of error.

Since we did not find a way to use *make*, two sets of compilers is the only solution we found to use Ninja.
After loading the modules, we tried two different build versions of ParaView: one with the qt module and one without.

```
cmake -GNinja -DPARAVIEW_ENABLE_PYTHON=ON -DPARAVIEW_USE_MPI=ON
      -DPARAVIEW_ENABLE_CATALYST=ON -DPARAVIEW_BUILD_QT_GUI=OFF ..

cmake -GNinja -DPARAVIEW_ENABLE_PYTHON=ON -DPARAVIEW_USE_MPI=ON
      -DPARAVIEW_ENABLE_CATALYST=ON ..
```

We compiled the code with both versions and found the same result: random errors. With "random errors" we mean that two consecutive tests of the same code with the same input can raise different errors. In addition to that, sometimes adding a *printf* in the adaptor code generates an error with hdf5 that maybe is linked with the different intel module version explained above. To isolate the problem we commented all the CoProcess function and gradually we uncommented it. At this stage the function that raise the error seems to be:

```
Processor->CoProcess(dataDescription);
```

We were not able to fix the error and we had to stop tring due to lack of time.
We suggest two possibilities that should fix the errors. The first one is building with the same intel module ParaView and the code. The second one is building ParaView for a headless machine (see Appendix A). This second suggestion should be easier, compared to Licallo, since llvm and qt libraries are available as modules.

For clarity the following it the slurm file used:

```
#!/bin/bash
#MSUB -r run             # Request name
#MSUB -n 8               # Number of tasks to use
#MSUB -c 1               # Number of threads per task to use
#MSUB -T 700            # Elapsed time limit in seconds
#MSUB -o RUN_job.out     # Standard output. %I is the job id
#MSUB -e RUN_job.err     # Error output. %I is the job id
#MSUB -m scratch,work
#MSUB -q xlarge          # rome, skylake, xlarge
#MSUB -Q test            ###normal ###test  ### queue
#MSUB -A gen12428
#MSUB -X

echo "Running on: $SLURM_NODELIST"
echo "SLURM_NPROCS=$SLURM_NPROCS"
echo "SLURM_NTASKS=$SLURM_NTASKS"
echo "SLURM_NTASKS_PER_NODE=$SLURM_NTASKS_PER_NODE"
echo "SLURM_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK"
echo "SLURM_NNODES=$SLURM_NNODES"
echo "SLURM_CPUS_ON_NODE=$SLURM_CPUS_ON_NODE"

date > start
ccc_mprun -Xfirst -n $SLURM_NPROCS ./iPIC3D
             ./testExosphere3D_yesAll_small.inp
#mpirun -n $SLURM_NPROCS ./iPIC3D ./testExosphere3D_yesAll_small.inp
date > end
```