Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

In-situ Visualization: State-of-the-art and Some Use Cases

Marzia Rivi[a,*], Luigi Calori[a], Giuseppa Muscianisi[a], Vladimir Slavnic[b]

[a]*CINECA, via Magnanelli 6/3, 40033 Casalecchio di Reno, Italy*
[b] *Scientific Computing Laboratory, Institute of Physics Belgrade, University of Belgrade, Pregrevica 118, 1108, Belgrade, Serbia*

Abstract

In this paper we present an investigation about techniques and frameworks supporting in situ-visualization. With this term we mean that visualization is coupled with simulation and it occurs whilst the simulation is running. By coupling these together we can utilize the high performance computing for post processing, and we can circumvent the bottlenecks associated with storing and retrieving data in disk storage. Moreover it allows monitoring the simulation in-situ, performing not only visualization, but analysis of the incoming data as it is generated so that the simulation may be stopped or modified, thereby conserving CPU resources. In particular we have tested two techniques, by exploiting different visualization tools on two applications. The first one is the astrophysics code Pluto instrumented by using a ParaView plug-in called ICARUS, the second one is the neural simulator code BrainCore instrumented by using a library of VisIt.

1. Introduction and goals of the project

The scientific community is presently witnessing an unprecedented growth in the quality and quantity of data coming from simulations and real-world experiments. Moreover writing results of numerical simulations to disk files has long been a bottleneck in high-performance computing. To access effectively and extract the scientific content of such large-scale data sets (often sizes are measured in hundreds or even millions of Gigabytes) appropriate tools and techniques are needed. This is especially true for visualization tools, where petascale data size problems cannot be visualized without some data filtering, which reduces either the resolution or the amount of data volume managed by the visualization tool. In fact, the primary bottleneck of the traditional visualization pipeline (where several nodes of an HPC system are used for computation, all/some of them write the data produced on the storage, and a visualization application exploits a smaller number of nodes of a separate system to read the data and visualize them) is due to I/O. However it is not always meaningful to post-process a simplified version of the data. Moreover these actions are not always applicable and may still require a significant amount of I/O operations. Finally, in petascale problems, data reduction could become a problem since it introduces additional errors that are not in the simulated data. Therefore, massive parallel visualization will become an issue.

Another way to avoid bottlenecks associated with storing and retrieving data in disk storage is to couple computation and visualization. In-situ visualization libraries enable the user to connect directly to a running simulation, examine the data, do numerical queries and create graphical output while the simulation executes. It addresses the need of extreme scale simulation, eschewing the need to write data to disk. Moreover it allows steering the simulation, i.e. to analyze the incoming data so that the simulation may be stopped or modified, thereby conserving CPU resources. For this reason in-situ visualization is becoming of particular interest in the scientific community.

The goal of this project is to investigate the state-of-the-art of in-situ visualization and apply two solutions, provided by the most popular general purpose visualization tools, to two applications. Before testing in-situ solutions, we have also examined the visualization tools used in this work: ParaView [1] and VisIt [2]. In this paper we present the results of this investigation. First, a brief survey on the main techniques implementing in-situ visualization is presented in Section 2. Then an overview of the features of the two aforementioned

* Corresponding author. *e-mail address*: m.rivi@cineca.it.

visualization tools is reported in Section 3 and 4, respectively. Testing of the graphical features and use cases of in-situ visualization are also included. In particular, benchmark results of some features enabling MPI, with and without the usage of GPUs, are presented in Section 3.3 for ParaView and Section 4.3 for VisIt. The system used for these tests is the Linux Cluster PLX [3] provided by CINECA.

2. In-situ visualization: different approaches

In-situ visualization allows to couple a simulation with visualization in order to visualize/post-process data produced by the simulation whilst it is running. Different techniques can be considered:

- *Tightly coupled* [4]: visualization and computation have direct access to the memory of the simulation code, analysis algorithms run on the same nodes/machine as the simulation (Fig. 1a). The disadvantages of this approach are consequences of the fact that visualization now requires the same level of resources as are being used for data generation. In fact a potential scalability and performance problems can occur because enough memory per node to support both post-processing and simulation is required. Moreover the simulation must wait for post-processing to finish before being able to carry on computation.
- *Loosely coupled* [5]: visualization and analysis run on concurrent resources and access data over network (Fig. 1b). Two possible architectures can be chosen: a pull-driven approach asks the simulation for a new time step when post-processing is finished, while with a push-driven approach a new time step produced by the simulation creates a new post-processing operation. Since this technique requires separate resources, memory copies between simulation and post-processing hosts are necessary and data transfers are limited by the network capabilities.
- *Hybrid*: data is reduced in a tightly coupled setting and sent to a concurrent resource (Fig. 1c). Obviously this technique shares drawbacks of both previous solutions (to a lesser extent), but can also take all their advantages. It can be complex to set up/implement and depends on the user needs.

There is no a universal technique for in-situ visualization. The choice depends on scientist needs, on data produced and on environment used to run simulation and post-processing. Resuming, tightly coupled approach do not require data movement but it is synchronous, loosely coupled approach requires data movement but it is asynchronous, hybrid approach is to be evaluated. Further information on this subject can be found on [6] and [7].

Fig. 1. (a) Tightly coupled approach: visualization and computation shared the same resources. (b) Loosely coupled approach: visualization and analysis run on separate resources. (c) Hybrid approach.

Currently, the following open-source frameworks implement in-situ visualization:

1. ICARUS (***I**nizialize **C**ompute **A**nalyze **R**ender **U**pdate **S**teer*), a ParaView plug-in for steering and visualizing in-situ HDF5 output of simulation codes. It implements a loosely coupled and push-driven approach: the link between simulation and visualization is decoupled to have a shared memory mapped file in between. It minimizes modification of existing simulation codes but requires data written using HDF5 format (see Section 3.4).

2. *ParaView Co-processing* [8]. This package allows to instrument parallel simulations to communicate to an outside visualization tool and generate images in parallel by the visualization code tightly coupled to the simulation. It has two main parts: an extensible co-processing library designed to be flexible enough to be embedded in various simulation codes with relative ease, and configuration tools for co-processing configuration that is important for the users to be able to configure the co-processor using graphical user interfaces that are part of their daily work-flow.
3. *VisIt libsim*. Another library implementing a tightly coupled approach that allows to instrument a simulation code so that VisIt can connect to it and access its data directly as though the simulation was a VisIt compute engine (see Section 4.4).

In this project we have experimented with the first (CINECA) and the third (IPB) framework on real applications.

3. Visualization tool: ParaView

ParaView [1] is an open-source, multi-platform application designed to visualize data sets of varying sizes, from small to very large. The goals of the ParaView project include developing an open-source, multi-platform visualization application that supports distributed computational models to process large data sets. It has an open, flexible, and intuitive user interface. Furthermore, ParaView is built on an extensible architecture based on open standards. Under the hood, ParaView uses the Visualization Tool Kit (VTK) as the data processing and rendering engine and has a user interface written using the Qt cross-platform application framework. This tool is fully scriptable using the simple but powerful Python language. ParaView's data engine, called server manager, is fully accessible through the Python interface. All changes made to the engine through Python are automatically reflected to the user interface. ParaView can be run as a batch job. Additional modules can be added by either writing an XML description of the interface or by writing C++ classes. The XML interface allows users/developers to add their own VTK filters to ParaView without writing any special code and/or re-compiling.

ParaView runs parallel on distributed and shared memory systems using MPI. These include workstation clusters**,** visualization systems, large servers, supercomputers, etc and has been successfully tested on Windows, Linux, Mac OS X, IBM Blue Gene, Cray XT3 and various Unix workstations and clusters. ParaView uses the data parallel model in which the data is broken into pieces to be processed by different processes. Most of the visualization algorithms function without any change when running in parallel. ParaView also supports ghost levels used to produce piece invariant results. Ghost levels are points/cells shared between processes and are used by algorithms which require neighborhood information. It supports distributed rendering (where the results are rendered on each node and composited later using the depth buffer), local rendering (where the resulting polygons are collected on one node and rendered locally) and a combination of both (for example, the level-of-detail models can be rendered locally whereas the full model is rendered in a distributed manner). This provides scalable rendering for large data without sacrificing performance when working with smaller data.

The user interface is run on a separate computer using the client/server mode. In this way, users can have the full advantage of using a shared remote high-performance rendering cluster. The ParaView client is a serial application and is always run with the `paraview` command. The server is a parallel MPI program that must be launched as a parallel job. There are two modes in which it is possible to launch the ParaView server. In the first one, all data processing and rendering are handled in the same parallel job. This server is launched with the `pvserver` command. In the second mode, data processing is handled in one parallel job and the rendering is handled in another parallel job launched with the `pvdataserver` and `pvrenderserver` programs, respectively. The point of having a separate data server and render server is the ability to use two different parallel computers, one with high performance CPUs and the other with GPU hardware. However, splitting the server functionality in two necessitates repartitioning and transferring the data from one to the other. This overhead is seldom much smaller than the cost of just performing both data processing and rendering in the same job. Thus, it is recommended on almost all instances simply using the single server.

ParaView is also used as a server of a web application aimed to provide a collaborative remote web interface for 3D visualization (ParaViewWeb). Some of the web samples are fully operational clients that allow to create remote web visualization as well as joining previously created ones with fully interactive user interface for building complex data processing and visualization.

3.1. Graphical features

A large variety of data types is supported: structured (uniform rectilinear, non-uniform rectilinear, and curvilinear grids), unstructured, polygonal, image, multi-block and AMR data. All processing operations (filters)

produce datasets. This allows the user to either further process the result of every operation or the results as a data file. On the other hand, this could lead to large memory requirements. A brief summary of filters provided by ParaView is the following.

- A sub-region of a dataset can be extracted by cutting or clipping with an arbitrary plane (all data types), specifying threshold criteria to exclude cells (all data types) and/or specifying a VOI (volume of interest - structured data types only). It is also possible to extract a subset of the input data by using a selection/subset/surface filter. This feature could be extremely interesting, since it allows to focus on a small sub-region of a large dataset, allowing the user to cut-out and retrieve only that part of data he is actually interested in. Data can also be extracted over time (including statistical information such as minimum, maximum and standard deviation). Also this feature is extremely meaningful, a large part of the data produced by simulations being time variant.
- Contours and iso-surfaces can be extracted from all data types using scalars or vector components. The results can be colored by any other variable or processed further. When possible, structured data contours/iso-surfaces are extracted with fast and efficient algorithms which make use of the efficient data layout.
- Vectors fields can be inspected by applying glyphs (arrows, cones, lines, spheres, and various 2D glyphs) to the input dataset, or by generating streamlines using constant step or adaptive integrators.
- New variables can be computed using existing point or cell field arrays (array calculator). A multitude of scalar and vector operations are supported. This is an important feature since often “observable” quantities are derived from the computed and stored ones.
- The volume rendering (VR) is available for unstructured data sets and image data (i.e. uniform rectilinear grid). The other structured data sets must be first converted into unstructured ones by using the *Tetrahedralize* filter. For image data, the VR algorithms are either CPU-based (fixed point) or GPU-based. For unstructured data set the VR algorithms are Projected tetrahedra, HAVS, Bunyk Ray Casting, ZSweep.
- Advanced data processing can be done using the Python Programmable filter.

A variety of file formats is provided including, first of all VTK. However, there is no a native reader of the HDF5 format [9], but it is possible to read this format by producing an XDMF (eXtensible Data Model and Format [10]) description of the HDF5 file. Furthermore, since ParaView is open source, the user can easily provide his/her own readers and writers.

3.2. Benchmarks

ParaView benchmarks are performed using the client-server modality on different nodes of the CINECA cluster PLX [3]. More precisely it is an IBM iDataPlex DX360M3 made of 274 compute nodes, each containing 2 NVIDIA® Tesla® M2070 and 2 Intel(R) Xeon(R) Westmere six-core E5645 processors. Moreover it has 6 RVN nodes for pre and post-processing activities, supporting DCV-RVN remote visualization software of IBM. PBS scheduler is used for submitting batch jobs.

CINECA group run the visualization tool on an RVN node equipped with IBM E5540 Intel(R) Xeon(R) CPU and Nvidia QuadroFX 1500. Each session is opened by establishing a VNC client/server connection with the RVN node. From the RVN node the paraview client is launched, while two compute nodes (each of them has 12 cores and 2 GPUs) are reserved for launching the ParaView servers. PBS scripts, describing the resource configuration (#cores, #GPU enabled, ...), has been prepared for the batch submission of the parallel pvserver sessions.

The dataset used for testing the performance of ParaView is a uniform rectilinear grid 512x512x512 contained in a HDF5 file. By writing a python macro, it has been measured the execution times of reading and of the following operations (the corresponding column number in Table 1 is indicated aside):

- reading: open the data file and perform the GPU-based volume rendering (c.2);
- zoom the image, visualizing 12 frames, by using the GPU-based volume rendering (c.3);
- rotate the image, with 40 time steps, by using the GPU-based volume rendering (c.4);
- zoom the image, visualizing 12 frames, by using the 'Fixed point'-based volume rendering (c.5);
- rotate the image, with 40 time steps, by using the 'Fixed point'-based volume rendering (c.6);
- compute a contour plot, with 200 points, and rotate the contour plot image, with 200 time steps (c.7);
- zoom the contour plot image, visualizing 12 frames (c.8);
- compute a list of contour plot, visualizing each of them (c.9).

For each operation we considered the following configurations:

- single cpu, where client and server run on the same RVN node;
- single node: 2, 4, 6, 8, 10, 12 pvservers run on the same compute node, enabling both the GPUs available in

the node for the GPU-based operations ;

- internode: 16 and 24 pvservers run on 2 compute nodes, enabling both the GPUs available on each node for the GPU-based operations.

Table 1. ParaView benchmarks on PLX Linux Cluster. Times are measured in secs.

Number of cores	Reading	Zoom VR_GPU	Animate VR_GPU	Zoom VR_noGPU	Animate VR_noGPU	Animate Contour	Zoom Contour	Loop Contour
1	3.2675	20.342	87.566	118.91	407.263	44.596	0.9882	88.128
2	3.7194	10.421	40.368	54.593	158.825	29.693	0.6145	50.068
4	3.8022	6.2596	23.566	38.672	120.943	32.127	0.7958	25.883
6	4.1879	1.4661	5.0248	25.514	103.496	33.300	0.8419	19.921
8	9.1062	1.5122	5.4592	19.989	75.6850	33.060	0.8430	14.622
10	9.6323	1.5309	5.7772	25.276	78.8909	35.057	0.8657	18.739
12	12.714	2.0876	7.9678	18.882	66.3318	35.703	0.9119	13.047
16	6.1404	1.2821	5.2407	11.313	45.3866	32.372	0.8383	9.4213
24	6.9719	1.5147	6.0448	9.5568	36.5775	33.491	0.8835	7.6140

Results are listed in Table 1. The numbers are in accordance with the underlying visualization computational organization: with the current benchmark data, ParaView is likely not able to exploit parallel reading (see Fig. 2a), so there is not much gain in this stage of computation, much better result would likely be obtained by splitting the data in chunks. The next columns refer to the timing of different rendering algorithms that try to simulate a user that interactively change the view point; two different camera models have been used: an animation around the object and a zoom-in: in the first the "screen size" of the object remain more or less constant while in the second it varies considerably. Regarding volume visualization algorithms, the CPU version is computationally expensive but shows a quite good scaling, but nevertheless, results confirm that GPU acceleration is relevant and, as expected, does not benefit much of an increase in CPU available (Fig. 3): the maximum performance is reached at around 6 processor (and two GPUs). We could interpret that with this data size, volume rendering on our single node is optimal. The latest column show the two stages of a contour type visualization, that could be a representative of a typical data flow pipeline where there is a data filtering stage (contour extraction) followed by a session of interactive polygon rendering (animate and zoom contour). The polygon rendering phase does not scale at all, showing that a single modern GPU is able of handling this data polygon size, the data filtering phase (CPU only) show instead a quite good scaling in CPU number (Fig. 2b).

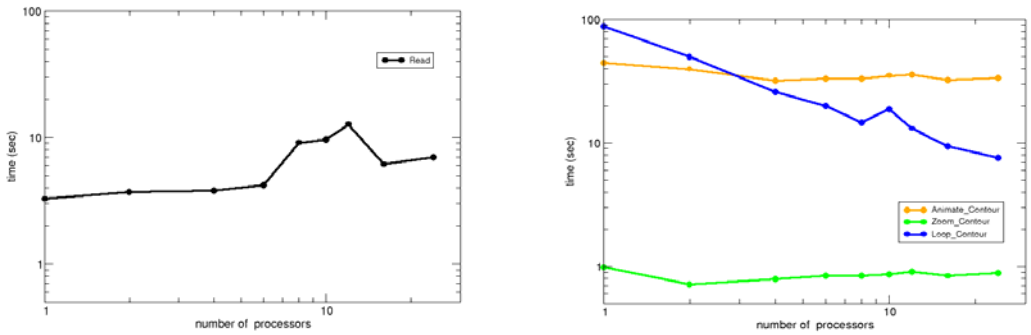

Fig. 2. (a) XDMF reader scalability. (b) ParaView performances for contour plot operations.

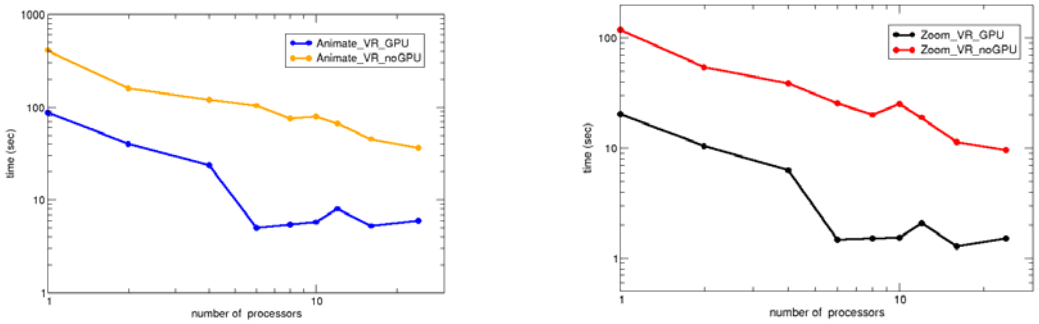

Fig. 3. (a) Comparison of the performance of rotation and volume rendering algorithm with and without GPU enabled. (b) Comparison of the performance of the zoom and volume rendering algorithm with and without GPU enabled.

3.3. The ICARUS plug-in

In principle, in order to monitor and analyse data coming from an HPC application at runtime you need to define an "interface" between the simulation itself and a post-processing software. Since the widely known HDF5 I/O library [9] already offers a modular mapping of file contents to storage, allowing the user to use different methods (drivers) for reading and writing data, the developers of the ICARUS plug-in have generated a new parallel virtual file driver for HDF5 called Distributed Shared Memory (DSM) driver [11]. It allows the transfer of data in parallel between two different codes, or between a simulation and a post-processing application, such as a visualization tool. This new driver has the capabilities of both *core*[a] and *mpio*[b] drivers provided in the standard HDF5 library. Moreover, it is able to transparently send data across a network in parallel to a distributed shared memory buffer allocated on different nodes of the same machine, or even of a completely different machine but accessible via the network. Therefore, the DSM driver allows two applications to be coupled together by using a shared file in memory as a common resource to which both may read/write and thereby control/steer one application from the other [12]. The DSM driver uses a client/server model where, generally, the simulation writing the data is the client and the set of (post-processing) nodes receiving the data is the server. The communication between such "client" and "server" can be based on sockets or on the MPI layer, which can also take advantage of the RMA operations provided by MPI.
The usage of the DSM driver to perform only in-situ visualization is really easy because you only need to link the H5FDdsm library and to replace in the simulation code the call to the mpio driver

```
herr_t  H5Pset_fapl_mpio (hid_t fapl_id, MPI_Comm comm, MPI_Info info)
```

with the call to the *dsm* one:

```
herr_t  H5Pset_fapl_dsm (hid_t fapl_id, MPI_Comm comm, void *dsmBuffer).
```

The data sent into the HDF5 driver by the simulation are automatically redirected to the DSM which interfaces ParaView, thanks to the ICARUS plug-in. In fact, this plug-in integrates the DSM host and several readers which accept HDF5 data, including a reader based on XDMF. Moreover the generation of XML descriptions of the data, required by XDMF to create topological connectivity from the raw HDF5 arrays, is provided. The plug-in checks for data ready synchronization messages and automatically updates the active pipelines in the ParaView GUI so that the visualization and any user generated analysis automatically track the ongoing simulation.

Regarding computational steering, both the simulation and the visualization can read and write datasets to the shared virtual HDF5 file layer, which allows the user to read data representing any aspect of the simulation and modify it using ParaView pipelines, then write it back to be re-read by the simulation (or vice versa). This allows not only simple parameter changes, but complete re-meshing of grids, or operations involving regeneration of field values over the entire domain to be carried out. To support bidirectional read/write access, the DSM driver is able to restrict access to the file from one side, whilst the other is using it, by using a locking mechanism. Moreover it provides a way to signal to the other side that new data has been written. Assuming that the

[a] *Memory based, it allows files to be written in memory instead of disk.*
[b] *It uses MPI-IO to write files in parallel and gives support to the user for collective/independent I/O operations.*

simulation makes regular/periodic writes to the file, there are two principal modes to proceed after a write operation: the simulation may wait for its data to be further processed and some new commands or data to be returned (*wait mode*), or it may continue making a quick check to see if anything has been left for it to act upon whilst it was calculating (*free mode*). The second one is entirely asynchronous and it is responsibility of the developer to ensure that the simulation can pick up data at a convenient point of the calculation by using a "manual signalling". Both the simulation and the host/server side may obtain the file access write and signal that the file is available for reading by using to the following procedure:

```
(1)   herr_t  H5FD_dsm_set_mode (unsigned long flags, void * dsmBuffer)
(2)   herr_t  H5FD_dsm_server_update (void * dsmBuffer)
```

Setting (1) with a manual update flag disables the automatic signal and (2) must be called when the DSM server switch signal is required. Therefore, in the free mode, a simulation whose calculation loops indefinitely issuing write commands, emits an update signal (via a call to (2)) whenever it has completed a step (and closes the file) and then immediately continues calculation on the next iteration. It may check for new commands/data at any time it reaches a convenient point in its algorithm where new data could be assimilated without causing a failure. The steering side meanwhile, receives the update command and immediately opens the file to read data and perform its own calculations. At this point, the steering application is post-processing time step T whilst the simulation has begun computing T+1 (assuming that we are talking about a simulation that iterates over time). Whilst the simulation is calculating, the steering side is free to perform analysis, modify parameters and write new data to the file. Usually, there will be a fixed pipeline setup in advance to slice, contour etc. and render the data as soon as an update signal is received. The user may interact with the GUI to modify the pipeline, change parameters and select outputs from it to be written back to the file. The steering API is based on such signal exchange between DSM server and simulation.

When using the DSM driver for visualization only, the connection is initialized during the HDF5 file creation alone, in the steering case it is necessary to add another initialization call (`H5FD_dsm_steering_init`) so that the first steering command exchange can be processed as soon as the simulation starts. Once the environment is initialized, the function `H5FD_dsm_steering_update` allows the user to get and synchronize steering commands with the GUI at any point of the simulation, although generally these commands will be at the immediate start and end of the main simulation loop. The functions `H5FD_dsm_steering_scalar_get/set` and `H5FD_dsm_steering_vector_get/set` allow the writing of scalar and vector parameters respectively, while `H5FD_dsm_steering_is_set` checks their presence in the file. Other functions provided by the steering library are listed in [13].

In conclusion, whereas live visualization does not require any modifications of the simulation code except setting HDF5 to use the DSM driver, simulation steering requires a higher degree of intrusion in the code. In general, an arbitrary HPC code can easily make use of the steering API described above. Once linked to the H5FDdsm library, the application doing an update of the steering orders automatically gets the new parameters or arrays passed to the file through the user interface provided by ICARUS.

To avoid the problem of manually customizing the GUI for each application to be steered, a significant portion of the work goes also into the creation of a XML template describing the outputs from the simulation, the parameters which may be controlled, and the inputs back to it. These templates are divided into two distinct parts: one called *Domain* describing the data for visualization only, and one called *Interactions* defining the list of steering parameters and commands one can control. The ICARUS plug-in generates from this template a complete (in memory) XDMF file with all the information about data precision and array sizes and a ParaView GUI panel (*DSMViewerPanel*). This panel is composed of three sections: one called *Basic Controls* containing the HDF5 DSM settings, the XML template settings and the data loaded in the DSM and built from the domain template; one called *Advanced Controls*, which allows to save images and export data; and finally one called *Steering Controls* containing the basic pause, play commands, and the used defined steering parameters (see Fig. 4).

CINECA group tested in-situ visualization and computational steering by using the instrumented version of the code Gadget2 [14], kindly supplied from the developers of the framework investigated (see [12]). Both the simulation code and ParaView were installed on PLX and they ran as separate MPI jobs using different number of cores. First, the pvserver was launched in batch on 2 compute nodes dedicated to visualization and then it was connected to the ParaView client (running on the RVN node) . Then, the ICARUS plug-in was loaded on both the pvserver and the client. After the DSMViewerPanel has appeared, on the Basic Controls panel the host was set as "default" and the inter-communicator as "MPI", because both the simulation and ParaView ran on the same machine. Finally, the "DSM data routing" was enabled to allow the data generated by the cosmological simulation to be shown on ParaView, and the path of the XML description template was defined. At the end of such preliminary operations, the HPC simulation was launched in batch and, after the job is started, the image

appeared on the ParaView panel. Moreover, on the Steering Controls panel, all the steering parameters were displayed and their values could be modified from this interface while the simulation was running. As expected, each modification of these parameters impacted on the results of the simulation.

3.4. Use case: PLUTO

In this section we present the experimentation of the previous technology on the astrophysical code PLUTO [15] developed at the University of Turin. It is entirely written in C and can run on either single processor machines or large parallel clusters through the MPI library. This application provides a simple user-interface based on the Python scripting language to setup a physical problem in a quick and self-explanatory way. Computations may be carried on either static or adaptive (structured) grids. The code embeds different hydrodynamic modules and multiple algorithms to solve the equations describing Newtonian, relativistic, MHD, or relativistic MHD fluids in cartesian or curvilinear coordinates.

As test case, a problem about “jets from young stellar objects” is considered. The numerical computations aim at investigating the formation of radiative shock waves in supersonic, axisymmetric magnetohydrodynamic (MHD) jets from Young Stellar Objects (YSOs). It is indeed widely accepted that these jets derive the bulk of their emission from post-shocked regions of gas where the large temperature and compression are responsible for the emission line spectrum commonly revealed by observations. The model investigated consists of a stationary jet model with a superimposed time-dependent injection velocity that produces a chain of perturbations eventually steepening into shock waves. This model is based on a comprehensive set of parameters, among which the perturbation period (PRD) and amplitude (AMP) play a fundamental role. A third parameter (BEG) controls the onset of perturbations since a pre-existing equilibrium state must have been reached. The equations of MHD are solved describing the evolution of density, momentum, energy density, magnetic field and a number of chemical elements including neutral hydrogen, helium and the first three ionization stages of C, O, N, Ne and S. Species are coupled through a comprehensive chemical network describing ionization and recombination processes. The computational domain is measured in AU and it is defined by $0 < r < 400$ and $0 < z < 1600$ with axis-symmetric boundary condition at $r = 0$ and injection condition at the bottom boundary ($z = 0$). Zero gradient conditions are applied on the remaining sides. The resolution amounts to 128 x 512 grid zones in the radial and vertical direction, respectively.

As well explained in the previous section, to perform only in-situ visualization, the HDF5 mpio driver was replaced with the new DSM driver in the simulation code. To perform steering in the free mode, the procedures for steering and for the manual signaling of the writing/reading of the HDF5 output file has been added, according to the following scheme:

```
. . .
H5FD_dsm_steering_init;
. . .
main loop:
H5FD_dsm_steering_update;
. . .
compute step;
. . .
if (writing on the HDF5 file):
  H5FD_dsm_set_mode(manual signaling);
  H5FD_dsm_steering_scalar_get("pertamplit");
  H5FD_dsm_steering_scalar_get("pertperiod");
  if (one or both these two parameters has been modified):
    update the corresponding variable of the simulation.
  H5FD_dsm_server_update;
endif
. . .
end loop;
```

Then an XML description file for the HDF5 datasets produced by the simulation was prepared by setting as steering variables both the perturbation amplitude (`pertamplit`) and the perturbation period (`pertperiod`). These parameters will be modified after a pre-existing equilibrium state has been reached:

```
 <Interaction>
  <DoubleVectorProperty
   name="pertamplit"
            command="SetSteeringValueDouble"
     label="PERT AMP"
     number_of_elements="1"
     default_values="70">
     <DoubleRangeDomain name="range" min="10" max="100" />
    </DoubleVectorProperty>

    <DoubleVectorProperty
     name="pertperiod"
     command="SetSteeringValueDouble"
     label="PERT PRD"
     number_of_elements="1"
     default_values="10.0">
     <DoubleRangeDomain name="range" min="1.0" max="70.0" />
    </DoubleVectorProperty>
 </Interaction>

<Domain>
   <Grid Name="Ysojet-grid">
     <Topology TopologyType="2DSMESH">
     </Topology>
     <Geometry GeometryType="X_Y">
     <DataItem Name="x">/node_coords/X</DataItem>
     <DataItem Name="y">/node_coords/Y</DataItem>
     </Geometry>

     <Attribute Name="Pressure" AttributeType="Scalar">
     <DataItem>/vars/pr</DataItem>
     </Attribute>
   . . .
    </Grid>
 </Domain>
```

The following figures show the results of the in-situ visualization and computational steering for the test case, by comparing two simulations starting from the same input file. The comparison shows the results at the same time step but with different values of the steering parameters (AMP and PRD), modified from the GUI of ParaView.

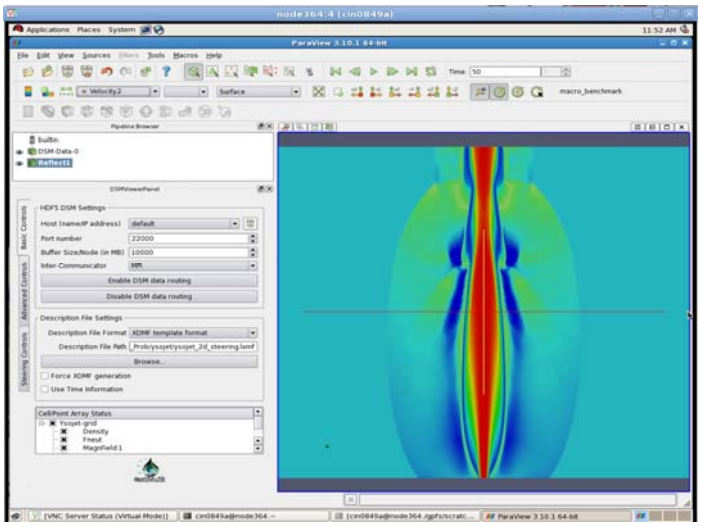

Fig. 4. Jet produced at the time t = 25.00. Steering parameters are still equal to the initial ones: AMP = 70.00, PRD = 10.00.

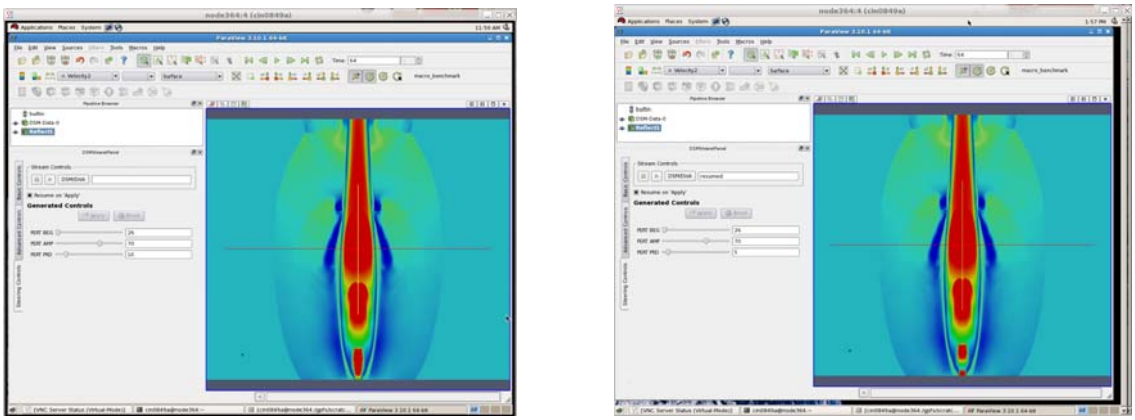

Fig. 5. Jet at the time t = 31.50. The cooling is started: chain of perturbations are produced from below. (a) Parameters are not changed. (b) The perturbation period PRD (3rd cursor of the GUI, moved by the user at t = 30.00) is decreased from 10 to 5. In fact, you can see that in the right image more perturbations are injected.

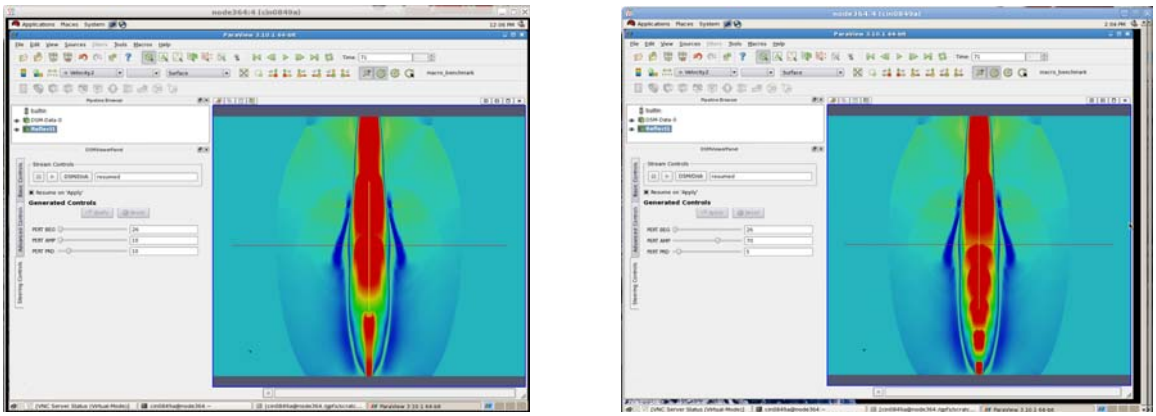

Fig.6. Jet at the time t = 35.00. (a) The perturbation amplitude AMP (2rd cursor of the GUI, moved by the user at t = 32.00) is decreased from 70 to 10, PRD = 10. (b) Parameters are not changed: AMP = 70, PRD = 5.

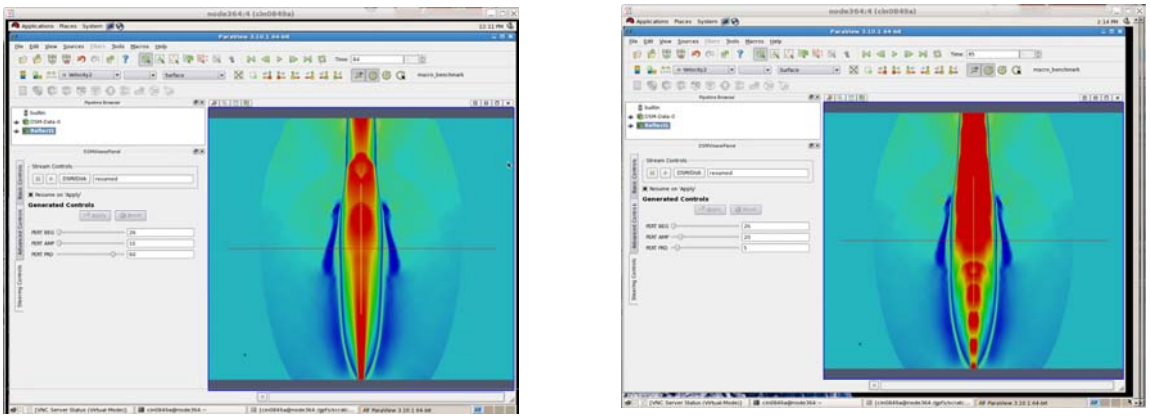

Fig.7. Jet at the time t = 41.50. (a) PRD is increased from 10 to 60, AMP = 10. (b) AMP is decreased from 70 to 20, PRD = 5.

During these tests no particular overheads were noticed. The only issue that can be mentioned is that both client and server must be up and running during the entire simulation. However this framework turned to be working well and not too difficult to install and use.

4. Visualization tool: VisIt

VisIt [2] is a free, open source, platform independent, distributed, parallel tool for visualizing and analyzing large scale simulated and experimental data sets. Target use cases include data exploration, comparative analysis, visual debugging, quantitative analysis, and presentation graphics. VisIt leverages several third party libraries: the Qt widget library for its user interface, the Python programming language for a command line interpreter, and the Visualization ToolKit (VTK) library for its data model and many of its visualization algorithms. VisIt maintains its own VTK fork. This software operates on UNIX (Irix, Tru64, AIX, Linux, Solaris), MacOS X (10.3, 10.4), and Windows platforms. VisIt employs a distributed and parallel architecture in order to handle extremely large data sets interactively. It can be controlled by a Graphical User Interface (GUI), through the Python and Java programming languages, or from a custom user interface that users can develop.

VisIt's architecture can be broken down into four major components. The first component is the GUI, which provides the user interface and menus. The Viewer component displays all of the visualizations in its windows and is responsible for keeping track of VisIt's state and for talking to the rest of VisIt's components. Both the GUI and the Viewer are meant to run on the local client computer so as to take advantage of the client computer's fast graphics hardware. The next two components can also be run on the client computer, but usually they are executed on a remote, parallel computer or a cluster where the data files are generated. The first such component is the database server, which is responsible for reading the remote file system and passing information about the files located there to the GUI on the local computer. The database server also opens a file to determine its list of variables and other metadata useful in creating visualizations. Finally, the compute engine is the component that actually reads the data files, processes them, and sends back either images or geometry to be drawn by the Viewer using the local computer's fast graphics hardware. Running VisIt in a distributed environment is simplified by providing users with a possibility to create host profiles containing information like remote username, number of processors, parallel launch method, etc. Host profiles are used to specify options for running VisIt's parallel compute engine on remote computers and usually the same host will have a serial host profile and several other host profiles with different parallel options.

Important additional feature is its in-situ capability. VisIt provides a library for in-situ visualization and analysis called libsim. It is used to instrument user's simulation code in order to allow VisIt to connect to it and access its data directly, as though the simulation was a VisIt compute engine.

4.1. Graphical features

VisIt has rich feature set for visualization of scalar, vector, and tensor fields with the ability to animate data, which allows users to see the time evolution of their data. Visualizations can be created from databases that are stored in many types of underlying file formats (almost all major formats used in computational science are supported), but also users are able to create VisIt plugins to make database readers for custom data formats. For its data presentation VisIt uses plots and operators. A plot is a viewable object, created from a database that can be displayed in a visualization window. An operator is a filter applied to a database variable before the compute engine uses that variable to generate a plot. VisIt provides several standard plot types, and in conjunction with different operators this enables visualization of data in different ways. Similar as for database readers, all of VisIt's plots and operators are plugins, so users can add new plots or operator types by writing their own plugins. The standard plots and operators perform basic visualization operations like:

- Pseudocoloring (pseudocolor plot) which maps a scalar variable's data values to colors and uses the colors to "paint" values onto the variable's computational mesh which results in a clear picture of the database geometry painted with variable values that have been mapped to colors.
- Contouring or isosurfacing with different types of plots and operators:
    - Contour plot that displays the location of values for scalar variables using lines for 2D plots and surfaces for 3D plots. In visualization terms, these plots are isosurfaces. VisIt's contour plot allows user to specify the number of contours to display as well as the colors and opacities of the contours.
    - Isosurface operator that extracts surfaces from 2D or 3D databases and allows them to be plotted. The isosurface operator takes as input a database and a list of values and creates a set of isosurfaces through the database. An isosurface is a surface where every point on the surface has the same data value.
    - Isovolume operator that creates a new unstructured mesh using only cells and parts of cells from

the original mesh and are within the specified data range for a variable. The resulting mesh can be used in other VisIt plots.

- Volume plot that uses a visualization technique known as volume rendering, which assigns color and opacity values to a range of data values. The colors and opacities are collectively known as a volume transfer function. The volume transfer function determines the colors of the plot and which parts are visible. This plot supports different type of volume rendering methods of which some are hardware accelerated (able to use fast GPU hardware) like splatting and 3D texturing and some are software based, like ray-casting which is slower but produces superior images comparing to other methods.
- Vectors visualization with vector and streamline plots. For example, vector plot displays vector variables as small glyphs that indicate the direction and magnitude of vectors in a vector field.
- Support for different operators for cutting, slicing and clipping of datasets.

VisIt also provides features for qualitative and quantitative visualization and analysis like expressions and queries. Variable expressions allow scientists to create derived variables using variables stored in the database. Expressions are extremely powerful because they allow users to analyze new data without necessarily having to rerun the simulation. It also supports a generalized query interface, which allows users to query derived quantities such as volume or surface area.

4.2. Benchmarks

VisIt benchmarks were performed using the client-server modality on different nodes of the PLX cluster as for ParaView (see Section 3.2). VisIt provides the useful feature of creating execution host profiles which enables simple initialization of the compute engines (for both serial and parallel engines running on different number of processors and nodes). This allowed to avoid any manual interaction with the PLX batch system (e.g. creating PBS scripts and submitting batch jobs by hand).

The dataset used for testing the performance of VisIt is an unstructured mesh contained in a SILO file of about 11MB. By using the Python script to control VisIt, the execution times of the reading and of the following rendering operations are measured (the corresponding column number in Table 2 is indicated aside):

- reading: open the data file and perform the GPU-based volume rendering (c.2);
- zoom the image, visualizing 12 frames, by using the GPU-based (Texture3D) volume rendering (c.3);
- rotate the image, with 40 time steps, by using the GPU-based (Texture3D) volume rendering (c.4);
- zoom the image, visualizing 12 frames, by using the software-based (Ray Casting) volume rendering (c.5);
- rotate the image, with 40 time steps, by using the software-based (Ray Casting) volume rendering (c.6);
- compute a contour plot, and rotate the contour plot image, with 40 time steps (c.7);
- zoom the contour plot image, visualizing 11 frames (c.8);

VisIt's scalable rendering feature (with hardware acceleration) was used in these tests. The dataset used for these benchmarks was unsuitable for test the loop contour part as done previously with ParaView. For each operation it has been defined the same configurations defined for the ParaView tests.

Results are shown in Table 2 and Fig. 8 and 9. Software rendering based on CPUs showed scaling with respect to increased number of CPUs while GPU based rendering showed no scaling because GPU acceleration was dominant and additional CPUs couldn't reduce the execution times (Fig. 9a). The last column (Zoom Contour) again shows no significant scaling because data processing (contour extraction) on this dataset was too short comparing to the rendering part done on GPUs. The dataset used in these tests is relatively small; therefore this can be a possible explanation why tests other than CPU-based volume rendering didn't show any significant scaling after 2 cores. Moreover an appropriate usage of data with the VisIt parallel version must be investigated.

Table 2. VisIt benchmarks on PLX Linux Cluster. Times are measured in secs.

Number of cores	Reading	Zoom VR_GPU	Animate VR_GPU	Zoom VR_noGPU	Animate VR_noGPU	Animate Contour	Zoom Contour
1	1.4425	32.2301	53.6479	46.6633	86.5450	41.2015	13.6576
2	0.7569	5.3358	19.3823	38.5732	72.0288	20.1435	5.5488
4	0.7291	5.3523	19.4053	33.3301	57.7587	28.7072	11.1379
6	0.7224	5.3322	19.3224	32.7100	56.0004	28.8330	9.5072
8	0.7198	5.3156	19.3730	30.0683	52.6964	29.0654	9.5996

10	0.7177	5.3501	19.4486	29.5682	51.0682	29.3900	9.5720
12	0.9856	5.3384	19.4815	29.4826	50.4101	29.5757	9.6710
16	0.6992	5.3441	19.4503	28.3104	47.5771	28.5908	8.6204
24	0.7135	5.3503	19.4353	27.5788	47.0309	26.4184	7.8814

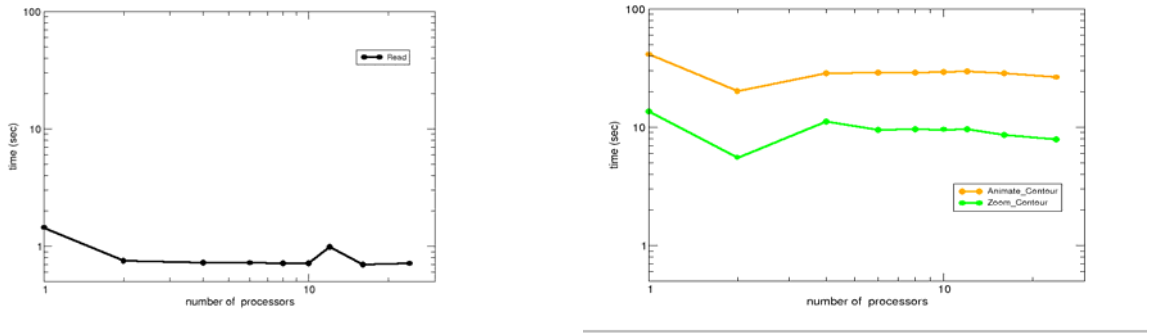

Fig. 8. (a) VisIt reader scalability. (b) Contour plot operations performances.

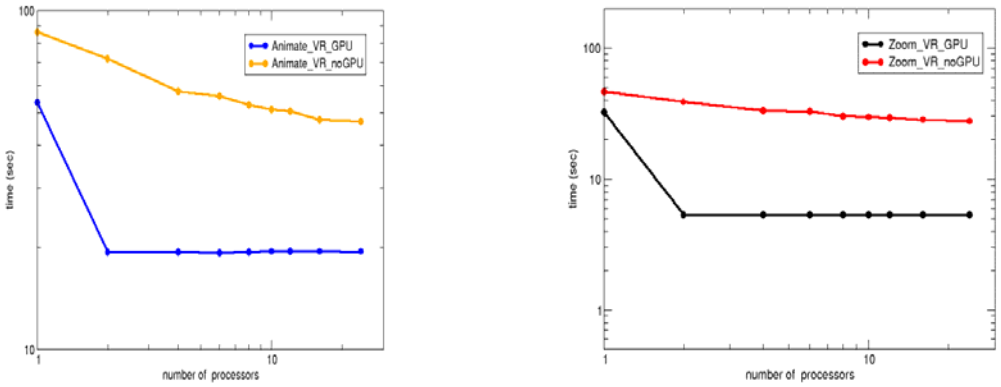

Fig. 9. (a) Comparison of the performance of rotation and volume rendering algorithm with and without GPU enabled. (b) Comparison of the performance of the zoom and volume rendering algorithm with and without GPU enabled.

4.3. Libsim library

VisIt live visualization approach is provided by its libsim library, which allows the visualization of simulation data in situ, thus avoiding the high costs of I/O associated with first writing and then reading the same data again. Libsim simulation instrumentation library can be inserted into a simulation program to make the simulation act in many ways like a VisIt compute engine. This library, coupled with some additional data access code that has to be written by a user and built into the simulation, gives to VisIt's data processing routines access to the simulation's calculated data without the need for the simulation to write files to a disk. The libsim library is responsible for listening to incoming VisIt connections, connecting to them, dynamically loading the runtime that allows the simulation to act as a VisIt compute engine, and responding to console user input or input from VisIt. An instrumented simulation may begin its processing while periodically listening for connections from an instance of VisIt using libsim. When libsim detects that VisIt wants to connect to the simulation to visualize its data, libsim loads its dynamic runtime library that contains the VisIt compute engine's data processing functions. Once the runtime is loaded, simulation connects back to VisIt's Viewer and requests for plots and data can be made as though the simulation was a regular VisIt compute engine. When a request for data comes in from VisIt's Viewer, simulation is asked to provide data via some data access code. Data access code consists of a set of callback functions that simulation must provide in order to serve data to VisIt. Data access code is written in the same language as the simulation program and acts as a connector that allows the runtime to access simulation's data.

VisIt can also export processed data back to simulations that implement write callback functions in their data access codes. Main steps for in-situ visualization with VisIt are described in the following subsections (more details can be found in [16]).

4.3.1. Building in libsim support

Libsim library comes in two flavors: *SimV1* and *SimV2*. The newer version of this library (SimV2) provides a function-based API for manipulating data and includes features not present in SimV1, such as support for Adaptive Mesh Refinement (AMR) meshes, Constructive Solid Geometry (CSG) meshes, material species, and many other features. Libsim is currently only available for Linux and MacOS X platforms.

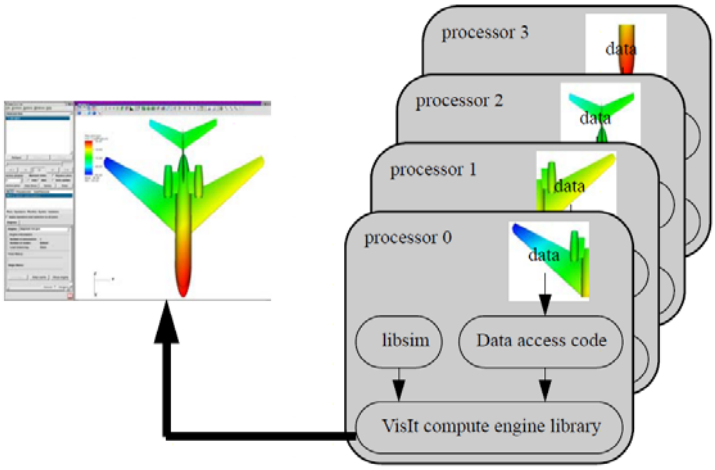

Fig. 10. Getting data to VisIt from an instrumented parallel simulation.

Simulation codes written in C or C++ must include `VisItControlInterface_V2.h` and `VisItDataInterface_V2.h` and has to be linked against the `libsimV2.a` library. Whereas, `visitfortransimV2interface.inc` must be included in FORTRAN simulation codes to assure that the compiler knows the names of the functions coming from libsim. Moreover a FORTRAN program must be linked against both `libsimV2.a` and `libsimV2f.a`.

4.3.2. Initialization

Common and recommended practice is to use a structure that represents the simulation's global state and which could also contain other data such as the mesh being simulated. Initialization functions are added to simulation's `main()` function. After calling optional functions, e.g. `VisItOpenTraceFile()` or `VisItSetDirectory()`, which respectively open a trace file containing a trace of all SimV2's function calls and set the path to the top level directory where VisIt is installed (e.g. `/path/to/VisItdir`), you must call the `VisItSetupEnvionment`() function when instrumenting a simulation code. It adds important visit-related environment variables to the environment, ensuring that VisIt is able to find its required plug-ins, etc.

Next step is to call the `VisItInitializeSocketAndDumpSimFile()`function, which initializes the libsim library and writes out a `.sim2` file to `~/.visit/simulations` directory in the user's home directory. A `.sim2` file is a small text file that contains details that tell VisIt how to connect to your running simulation.

For parallel programs libsim library requires periodic global communication to ensure that all processors service the same plot requests from VisIt's Viewer process. The usage of this library in a parallel simulation requires a small amount of extra setup. Process 0 needs to behave a little differently from the rest of the other processors because it communicates with VisIt's Viewer and then broadcasts necessary information to other processes.

4.3.3. Restructuring the main loop

It is highly recommended to implement the `mainloop` function which will be called from the simulation's

`main()`. It is possible to add calls to libsim into an existing simulation main loop using polling but it is not as clean as restructuring the main loop. If your simulation does not have a well-defined function for simulating one time step, it is strongly recommended that simulation is refactored so that code simulating one time step can be called from `mainloop` using either a single function or a small block of code.

Several functions from libsim should be placed in the `mainloop` function: `VisItDetectInput()` listens for inbound VisIt connections on a port allocated when libsim was initialized, and `VisItAttemptToCompleteConnection()`is responsible for dynamical loading the simulation runtime library which allows the simulation to perform compute engine operations, and connecting back to VisIt's viewer. In the event of a successful connection, the viewer and the simulation will be connected and the simulation will appear in the GUI's Compute Engines and Simulation windows. Another important function is `VisItProcessEngineCommand()` which is responsible for handling commands from the viewer. This function reads the commands coming from the viewer and uses them to make requests of the runtime, which ends up requesting and processing the data returned from simulation's data access code. When the simulation can no longer communicate with VisIt, it is important to call libsim's `VisItDisconnect()` function which resets libsim so it is ready to once again accept a new incoming VisIt connection.
In VisIt's parallel compute engine, the processor 0 communicates in any way with VisIt's viewer and broadcasts the requests for plots to all of the other processors so all can begin working on the request. For this type of simulations additional code is added to ensure that all slave processors also call `VisItProcessEngineCommand()`when needed.

4.3.4. Adding the data access code

User must write function to provide metadata to VisIt to let it know the names of the meshes and variables that are used for plotting. Then, functions that will pass mesh or data arrays to VisIt, should be added to the simulation code. VisIt data access functions are registered with libsim by using special callback registration functions defined in `VisItControlInterface_V2.h`. There is one callback registration function per data access function.

The first data access function should be the one that populates a metadata object. VisIt uses metadata to determine which meshes and variables are in a database and reading a database's (or simulation's) metadata is the first thing VisIt does when accessing it. The `SimulationMetaData` object contains lists of other metadata objects such as meshes, variables, curves, materials and simulation state metadata which must be exposed to VisIt client in order to plot any of simulation's data.

After the implementation of a function that returns metadata about the simulation's meshes and variables simulation, data access functions should be defined which return the actual mesh and variables data. Adding a mesh data access function means that user has to first write a function and call `VisItSetGetMesh()` to register it with libsim. VisIt deals with several mesh types and each mesh type has its own corresponding data object that represents its data. Accessing variable data requires registering a new data access function by calling the `VisItSetGetVariable()` function. This function returns `VariableData` object which is a simple wrapper of the array that user wants to return and includes an information about the array, including its type, number of components, number of tuples, and which program owns the array data (the owner flag indicates whether or not VisIt will be responsible for freeing the variable data array when it is no longer needed).

These data access function (for meshes and variables) must be registered after a successful call to `VisItAttemptToCompleteConnection()` so it should be done at the same time as the metadata callback is registered.

Another important aspect of in-situ visualization is simulation steering. VisIt allows simulation to provide the names of user-defined commands in the metadata object and, when such commands appear in a simulation's metadata, it influences VisIt to create special command buttons in the Simulations window. When user interacts with buttons in the GUI's Simulations window it causes calling simulation's command callback function, which then performs some action based on the name of the command being executed. These custom commands give user the opportunity to perform limited steering of a simulation within VisIt. Examples of simple simulation commands that user might want to expose in the metadata are the "run", "stop" and "step". With the "stop" button user could stop execution of the simulation, perform some simulation data analysis and then with "run" continue with the simulation execution. With the "step" button user could initiate execution of just one step of the simulation. For these purposes `VisItSetCommandCallback()` function is used.

4.3.5. Running the instrumented simulation

Libsim uses the `visit` command in user's path to determine where it can find the libsim runtime library. The libsim runtime library is linked to its dependent VisIt libraries and assuming libsim is able to locate its runtime library, the other VisIt libraries on which it depends should load without any problems. If VisIt is not in user's path it is necessary to export some VisIt environment libraries.

4.3.6. Connecting to an instrumented simulation

Connection to simulation is performed like opening regular data file, but user should browse to `~/.visit/simulations` the directory where `.sim2` files created by running simulation are stored on the simulation execution host. By opening the `.sim2` file VisIt initiates the contact with the user's simulation. The most common problem related to the connection between VisIt and the simulation, usually is due to a failure by libsim to automatically detect the VisIt environment and therefore is not able to load the simulation runtime library. The examination of libsim trace files should give more details about any problem. If the connection is successful user will be able to see the name of the simulation in the Compute engines window and the Simulations window and to proceed with data visualization and simulation steering.

4.4. Use case: BrainCore

VisIt's libsim in-situ approach has been used to visualize the data produced by the BrainCore code [17], which is one of the codes from the PRACE Preparatory Call project "Visualization of output from large-scale brain simulation" submitted by KTH. The project aims to visualize large-scale neuron simulations of 50 thousands to several hundreds of thousands. The proposed neural visualization application provides a tool to study and better understand activity resulting from large-scale neural simulations and to generate the corresponding animations. It could possibly also be used in real-time visualizations, including zooming into specific parts of a neural network.

BrainCore is an experimental code written in C++ and simulates an abstract Poisson spiking model of the global neocortical network of the brain. It is designed to obey what is known in terms of the overall structure of the long-range cortico-cortical connections. All communication is performed using MPI point-to-point functions.

IPB performed testing and analysis of the simplified BrainCore simulation code provided by the developer (Simon Benjaminsson, KTH). In cooperation with the developers, relevant parts of the code for efficient implementation of in-situ visualization using VisIt were identified. Convenient types of data representation were chosen as well as visualization activity of simulated neuron units in a more natural, comprehensive way.

During the instrumentation process additional data structures were added in order to control simulation execution and interaction with VisIt client. One of the added structures was `simulation_data` generally used in libsim instrumented simulations, which contains fields: `runMode`, that indicates the state of the simulation (running or not); `par_rank` and `par_size`, important for synchronizing BrainCore's MPI instances while interacting with VisIt; `cycle` related to the current simulation step; and some further fields specific for this program. Suitable data representations were chosen and mesh and variables were defined. Functions for retrieving simulation's metadata (`SimGetMetaData()`), defined mesh (`SimGetMesh()`) and variables defined over mesh that represent neural activity (`SimGetVariable()`) were added.

Changes in the `main()` function included the creation of the instance of the `simulation_data` structure, adding the VisIt initialization function `VisItSetupEnvironment()`, registering the callback functions for global communication:

```
VisItSetBroadcastIntFunction(visit_broadcast_int_callback);
VisItSetBroadcastStringFunction(visit_broadcast_string_callback);
```

then calling functions that set libsim to operate in parallel and set the rank of the current process within its MPI communicator:

```
VisItSetParallel(sim.par_size > 1);
VisItSetParallelRank(sim.par_rank);
```

and finally adding the function `VisItInitializeSocketAndDumpSimFile()`, that will be executed only by the process with rank 0.

`NetworkRun()`method of `Network` class was used as a simulation mainloop function (typical libsim in-situ approach), where all interactions with VisIt were defined and through which single steps of simulation were

called (`Simulate()` method of `Network` class). Function `VisItDetectInput()` was added to detect VisIt client input from the listen socket, client socket, or console, and switch block in which different actions were defined, depending on the output of the `VisItDetectInput()` function: to continue with the simulation (simulate one step more) if there is no VisIt input, to try to successfully connect to VisIt if that kind of attempt was detected, to respond to VisIt's request to perform a particular compute engine command, and finally to detect an error in VisIt interaction. It is important to say that only root MPI process (with rank 0) performs execution of `VisItDetectInput`() function and it broadcasts its output to all other MPI instances. In case of successful connection with remote VisIt client, functions that forward metadata to client and perform registering of functions for accessing the mesh and variables data are executed:

```
VisItSetCommandCallback(ControlCommandCallback, (void*)sim);
VisItSetSlaveProcessCallback(SlaveProcessCallback);
VisItSetGetMetaData(SimGetMetaData, (void*)sim);
VisItSetGetMesh(SimGetMesh,(void*)sim);
VisItSetGetVariable(SimGetVariable, (void*)sim);
```

The first function registers the `ControlCommandCallback()` function, which allows steering of the simulation through VisIt simulations window and Commands buttons from the Controls tab (like stopping the simulation, running the simulation, updating the plots, etc. `VisItSetSlaveProcessCallback()`sets the callback.

Testing of the instrumented code was performed on the CINECA cluster PLX, where control of simulation was demonstrated and in-situ visualization of neural units' data was performed.

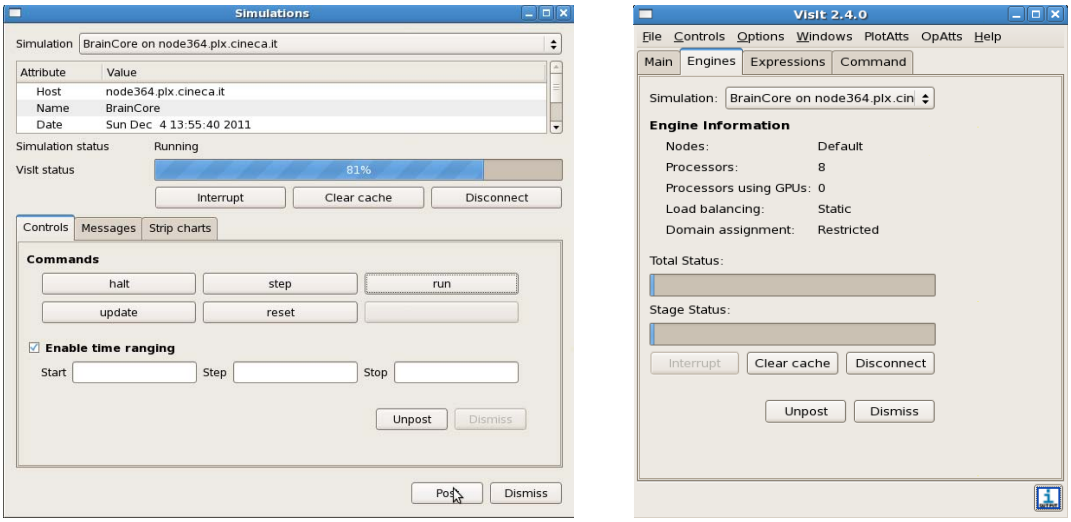

Fig 11. VisIt (a) Simulations window with control buttons. (b) Compute Engines window while it is connected to running BrainCore simulation.

5. Conclusions

Parallel visualization is a mature technology, but was optimized as a stand-alone process. It can run like a supercomputer simulation, but is limited by I/O. In-situ visualization is an attractive strategy to mitigate this problem, but requires an even stronger collaboration between the application scientists and the visualization scientist, and the development of a new family of visualization algorithms

The two visualization tools investigated, ParaView and VisIt, have strong similarities: they are both based on VTK as a general purpose scientific visualization toolkit (even if they use different version / patches), use Python as a scripting-glue language and they both implement their different user interface using Qt library.

Both tools try to hide the complexity of underlying data-flow VTK pipeline structure by providing functionality at a higher level (data loading, filtering and visualization algorithms). However they differentiate slightly regarding the target user: VisIt is a higher level tool, designed for scientists while ParaView is more flexible but need computer scientist's skills to be extended and customized. For example, Visit has a greater

number of pre-built data reader and volume visualization methods that are not immediately available in ParaView without extending it with plug-ins.

ParaView seems to be better optimized for GPU performance in Volume Rendering and also able to better automatically distribute single block data among available processor, while Visit apparently rely on parallel data reader for distributing data among processors. Nevertheless Visit is able to open more files “out of the box” while Paraview seems to need a “Visit plug-in” to access all VisIt available readers.

Regarding support for in-situ visualization, both tools implement a tightly coupled approach which requires a good knowledge of the simulation code to be instrumented. We tested only the one provided by VisIt on the BrainCore code. For ParaView we preferred to test a separate ICARUS plug-in which implements a loosely coupled approach. In fact, the flexibility of ParaView allowed other developers (from CSCS) to add another in-situ visualization technology besides the one already integrated in the tool. This new framework was tested on the astrophysics code PLUTO. The code was instrumented in order to do both in-situ visualization and steering. ICARUS turned to be less intrusive in the simulation code to steer applications. Moreover it does not depend on the underlying model – any type of simulation writing HDF5 files can use this framework. More details on this work can be found at the web page `https://hpc-forge.cineca.it/trac/viz-simula/`.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources [PLX, CINECA, Italy].

We thank Jerome Soumagne (CSCS), one of the developers of ICARUS, for his explanations and support in the usage of this plug-in. We also thank Jean Favre (CSCS) for his suggestions and support with the usage of VisIt.

References

1. A. Henderson, ParaView Guide, A Parallel Visualization Application. Kitware Inc., 2005. Web site: `http://www.paraview.org`
2. VisIt Users Manual, `https://wci.llnl.gov/codes/visit/1.5/VisItUsersManual1.5.pdf`. https://wci.llnl.gov/codes/visit/
3. IBM-PLX User Guide, `https://hpc.cineca.it/content/ibm-plx-user-guide`
4. N. Richartn, A. Esnard, O. Coulaud, Toward a Computational Steering Environment for Legacy Coupled Simulations. ISPDC’07.
5. H. Yu, C. Wang, R. Grout, J. Chen, K.-L. Ma, In Situ Visualization for Large-Scale Combustion Simulations. Computer Graphics and Applications, IEEE 30, 3 (2010) 45-57.
6. H. Childs, Architectural Challenges and Solutions for Petascale Postprocessing. Journal of Physics: Conference Series 78, 1 (2007) 012012.
7. H. Childs, Keynote Exascale visualization, EGPGV 2011.
8. `http://www.paraview.org/Wiki/CoProcessing`
9. The HDF group, Hierarchical Data Format Version 5 (2000-2001). `http://www.hdfgroup.org/HDF5`
10. `http://www.xdmf.org`
11. J. Soumagne, J. Biddiscombe, J. Clarke, An HDF5 MPI Virtual File Driver for Parallel In-situ Post-processing. In: Recent Advances in the Message Passing Interface, R. Keller, E. Gabriel, M. Resch, J. Dongarra, (eds.), Springer Berlin / Heidelberg, 2010, vol. 6305 of Lecture Notes in Computer Science (2010) 62-71.
12. J. Soumagne, J. Biddiscombe, Computational Steering and Parallel Online Monitoring Using RMA through the HDF5 DSM Virtual File Driver. Procedia Computer Science 4 (2011) 479-488.
13. J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, J-G. Piccinali, Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver. EGPGV 2011, 91-100.
14. V. Springel, The cosmological simulation code GADGET-2, Monthly Notices of the Royal Astronomical Society 364 (2005) 1105-1134.
15. A. Mignone, G. Bodo, S. Massaglia, T. Matsakos, O. Tesileanu, C. Zanni, A. Ferrari, A numerical code for computational astrophysics. Astrophysical J. Suppl., 170 (2007) 228-242.
16. Visit Tutorial In-Situ, `http://visitusers.org/index.php?title=VisIt-tutorial-in-situ`
17. S. Benjaminsson , A. Lansner A, Extreme Scaling of Brain Simulations, Jülich BG/P Extreme Scaling Workshop 2011, Juelich Technical Report.