Simulations go Live, a.k.a. *in-situ* visualization

Jean M. Favre, jfavre@cscs.ch, CSCS *(Swiss National Supercomputing Center)*, Visualization Software Engineer

L'écriture des fichiers des résultats de simulations numériques est depuis longtemps le goulot d'étranglement du calcul à haute performance. La visualisation *in-situ* permet à l'utilisateur de se connecter directement à une simulation en cours d'exécution, pour examiner les données et en faire des représentations graphiques. C'est donc un nouvel atout dans la course aux calculs de grande taille, en évitant le besoin d'écrire des données massives sur les disques. Mais, c'est aussi une technique facile d'utilisation pour des calculs de taille plus restreinte, pour laisser l'ingénieur se concentrer sur le développement de l'algorithme numérique, en rajoutant une interface graphique en très peu d'effort.

Writing results of numerical simulations to disk files has long been a bottleneck in high-performance computing. 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. Yet, it can also be used on the desktop by anyone wanting to concentrate on coding an algorithm, and adding a graphical user interface in a few button clicks.

The annual **Supercomputing conference** (SC10) has just closed its doors. Several machines around the world are now beyond the 1 petaFLOPS range **(one quadrillion floating point operations per second)**. Running simulations on these machines means using a computation distributed over 100,000 to more than several millions of processing elements. And writing results to disk files hits a major bottleneck. The access speed of disk drives is several orders of magnitude smaller than that of cpu-to-memory accesses. Future systems already in the design phase for the exascale era will only worsen this situation. (Read about the brutal facts of HPC at www.hp2c.ch/background/hpcfacts).

Thus, the need to eschew I/O to disk, in favor of *in-situ* visualization: visualize while the simulation is running, gaining direct access to the memory pointers of the simulation code, asking the simulation to do its normal iterative processing with regular checks to service visualization requests.

2010 is the first year that the Supercomputing Conference proposes tutorials in ***in-situ* visualization**. In fact, not just one, but two libraries in open-source form were demonstrated. The **ParaView package** proposes a co-processing toolkit [1] while the **VisIt package** demonstrated its *in-situ* support with the libsim library [2].

This article will present the VisIt library, because it is much more mature, and easier to operate. But the fundamental operating modes are similar in both libraries.

Simulation codes and I/O

Many scientific applications involve the solution of partial differential equations. These equations are discretized on a grid of cells or nodes and an approximation to the solution is generally found by iterating until a convergence threshold or when a maximum number of iterations is reached. What happens in between can be a long story. The programmer is faced with many challenges. Setting up the correct boundary conditions; iterating over the right data arrays; writing the data results in a format that is compatible with visualization tools. Given the availability of today's computing platforms, the programmer is encouraged to try bigger and bigger computing challenges, and will sooner or later move to parallel computing. With it comes the challenge of doing parallel I/O to disk, and to do it efficiently. This is still today where simulation codes are less advanced, exacerbating the I/O bottleneck discussed above. Writing I/O code has also often been the last and least fun process in implementing numerical simulations. Researchers prefer concentrating their coding efforts into the numerical parts akin to their discipline, leaving the I/O for last.

In the 1990s, scientific visualization has flourished as a post-processing activity. Running computer simulation consisted in doing batch-oriented computation on large *clusters* or supercomputers, followed by interactive visualization. This led to the development of many application-driven file formats for archiving, and to the necessity to develop writing and reading programs to encode, and later decipher the data file contents. The interaction with disk files has been a mandatory and often painful fact of scientific visualization, before one could even create the first image. To make things even worse, the visualization hardware is traditionally smaller, or even much smaller than the supercomputing platform first used for the computations, and the time spent reading data from disk files can be the major performance hit preventing interactive data exploration, impeding data discovery.

Instrumentation

What if one could directly visualize the progress of a simulation, with a live connection to the simulation code, being able to peek at any memory arrays and mesh structures, being able to confirm the correct simulation setup and iterations, without the need to save data to disk?

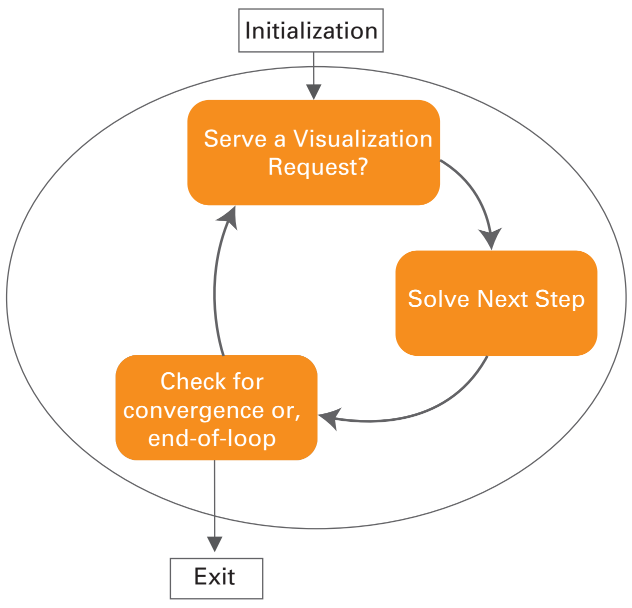

Initialization
Serve a Visualization Request?
Solve Next Step
Check for convergence or, end-of-loop
Exit

fig. 1 – the control flow loop of a simulation instrumented for in-situ visualization

Compilation and flow control

VisIt uses the basic client-server model, with a client running the GUI, and a parallel server [3]. The server runs an Engine Library where all the visualization algorithms are implemented. Running with an *in-situ* connection, consists in compiling and linking our simulation codes with the **libsim library** to gain access to **VisIt's Engine Library**. One then uses VisIt's client, the GUI component. Any visualization query available through VisIt's standard GUI is also available to the simulation. No previously defined visualization scenario must be encoded. At any time while the simulation executes, VisIt's GUI will be able to connect and disconnect from it.

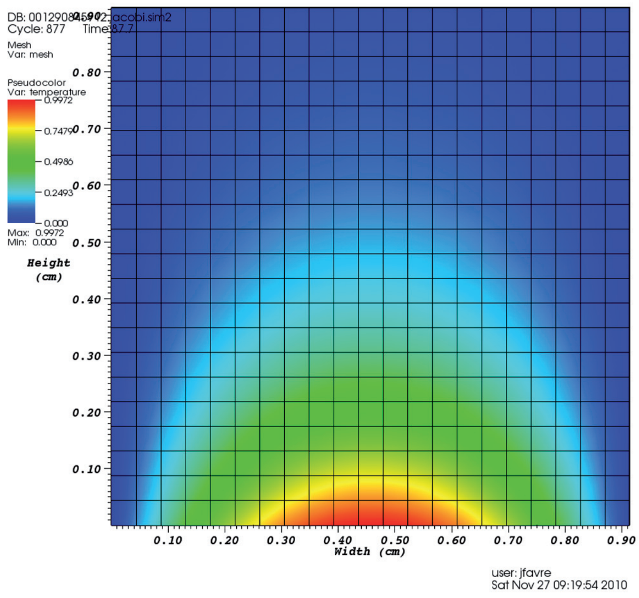


fig. 2 – pseudo-color display of temperature at timestep 877 during the execution of a Laplacian solver

Implementing the execution control of figure 1 might require some code re-organization, but the changes are usually small. Loops are usually found in the execution path of a simulation, and we only need to add a few control lines to allow the following:

- Establish the connection to the VisIt GUI.
- Receive and serve requests for data queries.
- Disconnect and let the simulation continue.

Our instrumentation of a FORTRAN95 simulation of a free-surface flow (FVRIVER) at CSCS required 68 lines of new source code. Not a big change in the main looping code!

Data Access

The main premise of *in-situ* visualization is to gain access to the memory contents of the simulation. Both C and FORTRAN simulations can be instrumented.

A second source code change to make is to enable read access to the pertinent data structures in the simulation code. All memory arrays can be advertized, enabling access to mesh and field variables, at any timestep, and for any parallel compute nodes participating in the simulation.

Meta-data information needs to be sent to the GUI. Meta-data are information about the mesh size, type and partitioning in the simulation, plus the number and type of variables available. This exchange of protocol with the GUI enables all the visualization techniques implemented for that type of data. For example, if a 2D rectilinear mesh with a variable called *temperature* is advertized, the user will be able to request a pseudo-coloring display of the surface, as well as iso-contour levels, histograms (etc…) of the scalar temperature.

Figure 2 shows such display. Source code for the Laplacian solver pictured here is available for your own testing [4].

Giving access to the temperature array (T) of the simulation above is done with few lines of code:

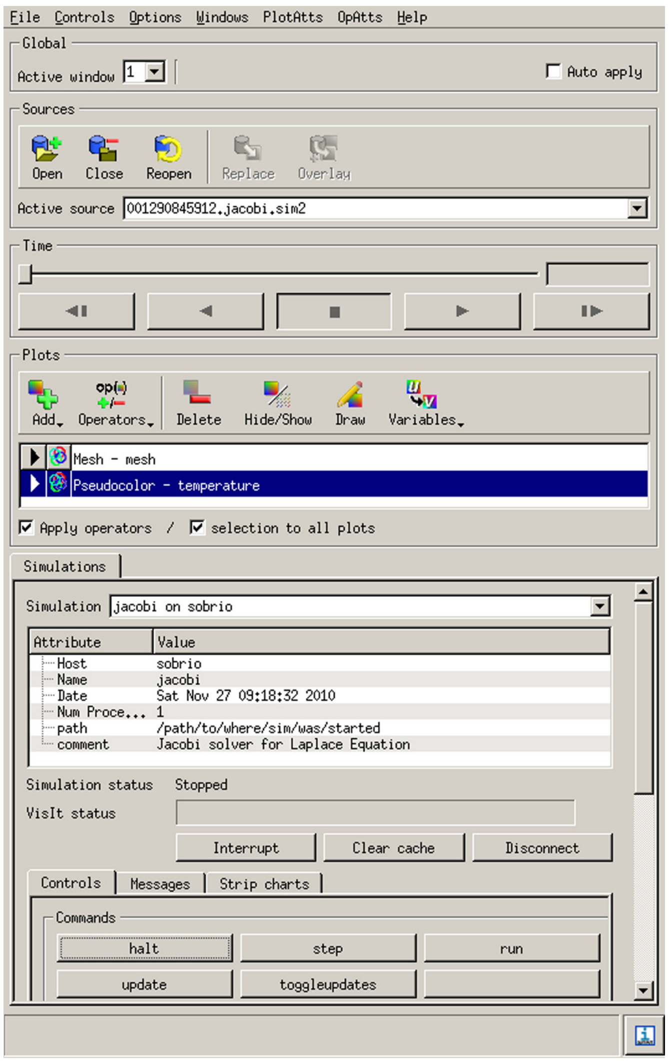
```
! FORTRAN code
allocate (T(XX, YY))
nTuples = XX * YY

visitvardatasetd(
  h, VISIT_OWNER_SIM, 1, nTuples, T)
```

```
// C code
float *T = malloc(XX * YY * 4);
nTuples = XX * YY;
VisIt_VariableData_setDataF(
 h, VISIT_OWNER_SIM, 1, nTuples, T);
```

The single most-important thing to notice is the flag **VISIT_OWNER_SIM** which indicates that the simulation code **owns** the memory pointer and is thus responsible for its deallocation. The visualization server component, loaded via a run-time `dlopen()` call when the connection is first established, is then free to use the pointer in read-only mode, to construct the visualization requested. This is the best scenario, thanks to a linear and compact array allocation, requiring no memory duplication. There will however be other cases, when an existing simulation code has a predefined memory usage which presents data in a more distributed - fragmented - manner. Just think of memory allocations dispersed across many `struct()` or C++ class members. In that case, the driver needs to gather the memory objects into a compact array allocated on purpose, and the visualization is said to own the memory pointer gathering the data. The VISIT_OWNER_COPY would distinguish this case.

Interaction with the simulation



Active window 1 ☐ Auto apply

Active source 001290845912.jacobi.sim2

Mesh - mesh

Pseudocolor - temperature

☑ Apply operators / ☑ selection to all plots

Simulation jacobi on sobrio

Attribute	Value
Host	sobrio
Name	jacobi
Date	Sat Nov 27 09:18:32 2010
Num Proce...	1
path	/path/to/where/sim/was/started
comment	Jacobi solver for Laplace Equation

Simulation status Stopped

VisIt status

Interaction with the simulation can be done through a command panel with buttons enabling iteration controls such as **next**, **update**, **run**, etc.

Besides visualization commands, a simulation code can also be instrumented to receive other types of inputs, such as parameters to influence the next steps in the simulation. The best demonstration is available in the Mandelbrot.C example of VisIt's source tree. In that demo, parameter inputs typed at the console, are used in future iterations to modify the number of levels and refinement ratio of the AMR grid used by the simulation.

Images

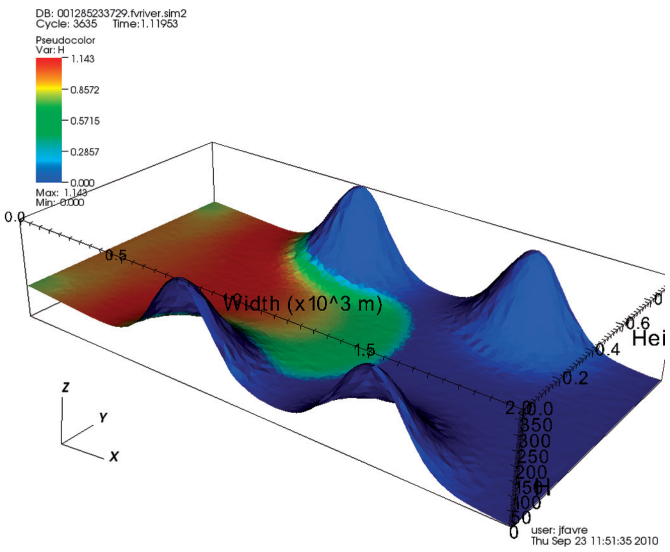
Let us not forget the original goal of visualization: to make pictures. Once connected to a simulation, with a set of visualization representations selected on-the-fly at a given timestep, we can instruct the Visualization server to save images to disk while the simulation iterates. We are thus replacing a possible enormous 3D transient archiving of results of raw data, with a sequence of well defined visualization images.

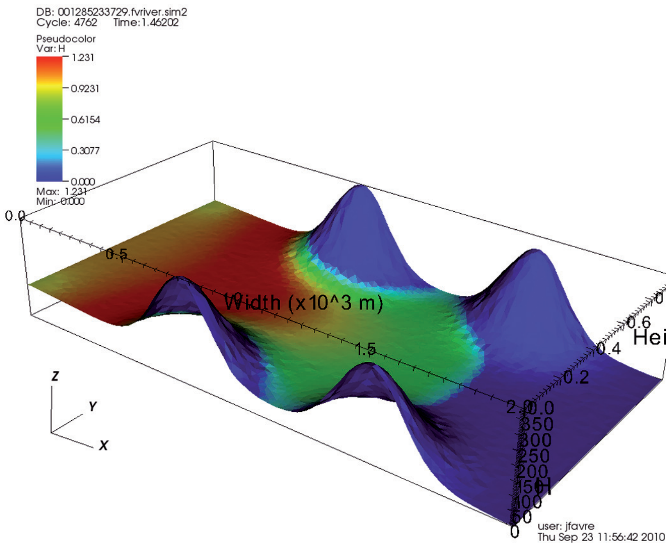
Conclusion

VisIt's *in-situ* visualization support, a mature interface to parallel FORTRAN and C codes, is an answer to one of HPC's fundamental road block towards increased performance at extreme scale. But it can also serve with modestly sized computations run on *clusters*, for anyone wanting to free himself from I/O and data formatting issues, to connect *live*, to a running simulation, and gain access to a myriad of visualization algorithms implemented and rigorously tested in VisIt's main visualization library. The author wishes to encourage any scientist writing his or her simulation code to install the open source VisIt (it is now far easier than it used to be), instrument their source code with compilation flags to conditionally compile the VisIt *in-situ* support, and then launch their simulation and connect the GUI with it.

References

[1] www.paraview.org/Wiki/SC10_Coprocessing_Tutorial;
[2] visitusers.org/index.php?title=VisIt-tutorial-in-situ;
[3] visitusers.org/index.php?title=500words;
[4] portal.nersc.gov/svn/visit/trunk/src/tools/DataManualExamples/Simulations/contrib/pjacobi/





Water height above the river bed at two different timesteps during an in-situ visualization connection with the solver FVRIVER of Matthew Cordery at CSCS