VisIt View package proposes a co-processing toolkit [1] while the Para-
two libraries in open-source form were demonstrated. The
flash informatique HPC at era will only worsen this situation. (Read about the brutal facts of
orders of magnitude smaller than that of cpu-to-memory access-
hits a major bottleneck. The access speed of disk drives is several
millions of processing elements. And writing results to disk files
writing results to disk. Later decipher the data file contents. The interaction with disk files
has been a mandatory and often painful fact of scientific visu-
alization, 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 inter-
active data exploration, impeding data discovery.

Simulation codes and I/O

Many scientific applications involve the solution of partial dif-
fential 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 maxi-
mum number of iterations is reached. What happens in between
can be a long story. The programmer is faced with many chal-

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?
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.

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 advertised, 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
allocate (T(XX, YY))
nTuples = XX * YY
visitvardatasetd(
    h, VISIT_OWNER_SIM, 1, nTuples, T)
```

```c
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 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**