CSCS

Swiss National Supercomputing Centre

In-situ Visualization

Dr. Jean M. Favre

Scientific Computing Research Group

13-01-2011

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

Outline

- Motivations
- How is parallel visualization done today
  - Visualization pipelines
  - Execution paradigms
  - Many grids or subsets on few processors
- A 2D solver with parallel partitioning and “patterns”
- *In-situ* visualization
  - Enable visualization in a running simulation
  - Source code instrumentation
  - Specify ghost-cells

Context

In-situ visualization is one action item in our HP2C efforts

The HP2C platform aims at developing applications to run at scale and make efficient use of the next generation of supercomputers. Presently this will be the generation of computing technologies available in 2013 timeframe.

Motivations

- Parallel simulations are now ubiquitous
- The mesh size and number of time steps are of unprecedented size
- The traditional ***post***-processing model "*compute-store-analyze*" does not scale because I/O to disks is the slowest component

Consequences:

- Datasets are often under-sampled on disks
- Many time steps are never archived
- It takes a supercomputer to re-load and visualize supercomputer data

"Visualization at Supercomputing Centers", IEEE CG&A Jan/Feb 2011

As we move from petascale to exascale…., favor the use of the supercomputer instead of a vis. cluster"

- Memory must be proportional to size and is $$$$$
- Aggregate flops instead of a parasitic expense
- Close coupling
- I/O

But,

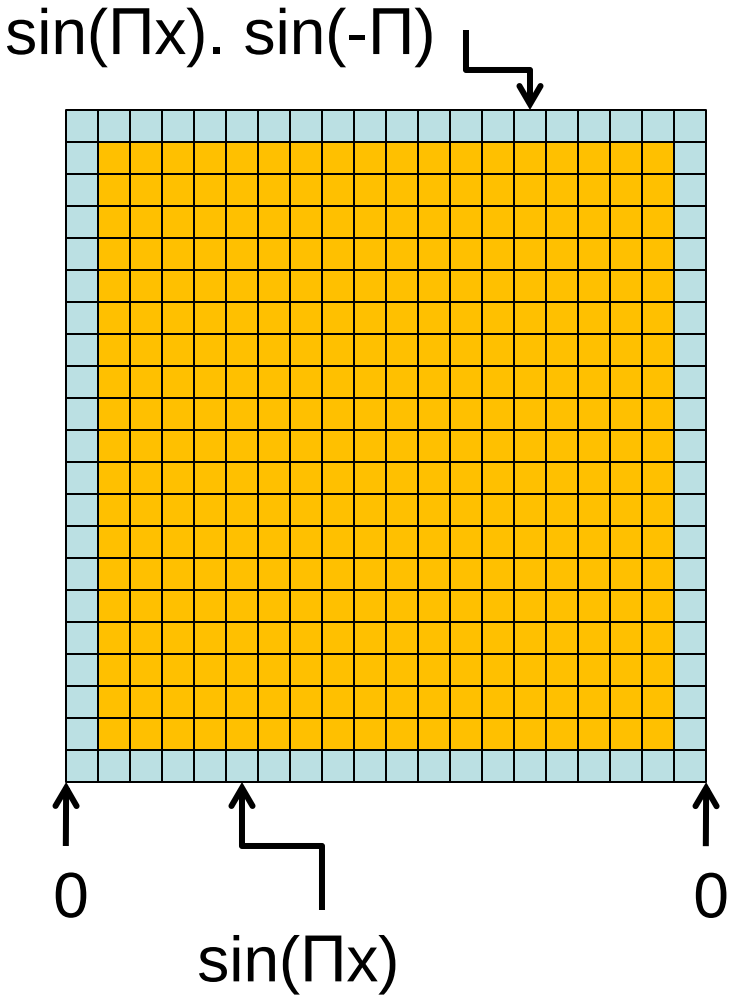
- Little memory per core

Solving a PDE and visualizing the execution

Full source code solution is given here:

- http://portal.nersc.gov/svn/visit/trunk/src/tools/DataManualExamples/Simulations/contrib/pjacobi/

A PDE with fixed boundary conditions

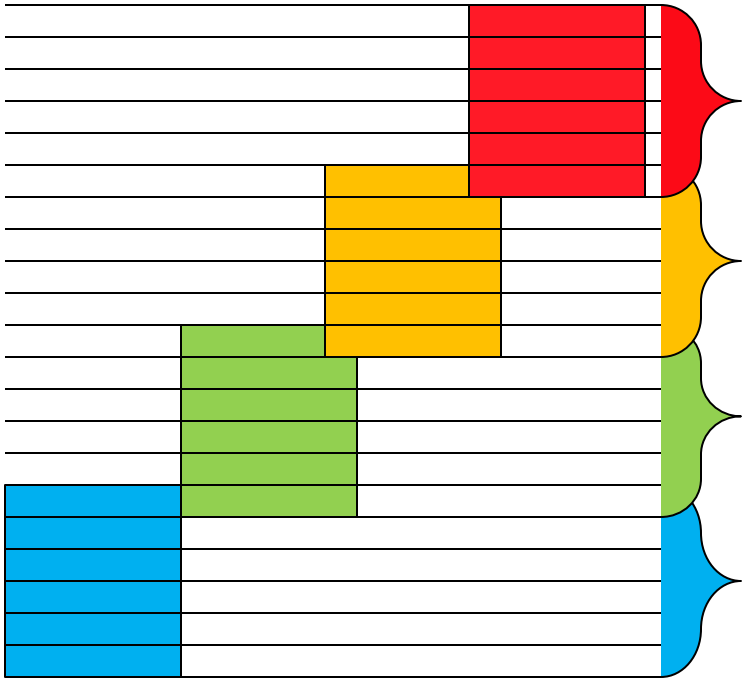


Update grid with solver

Fixed boundary conditions

Laplace equation: $\Delta u = 0$

2D grid partitioning and I/O pattern



BC init, or restart

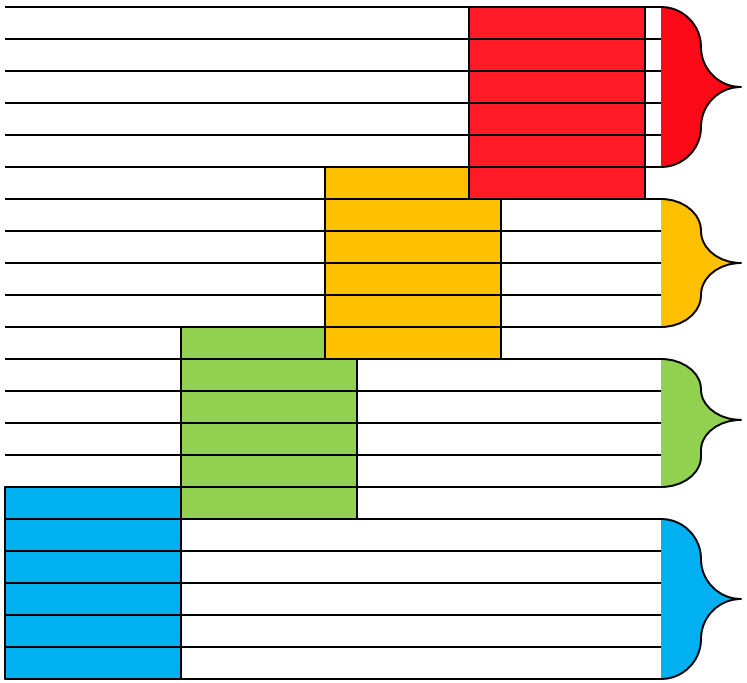
(mp+2) grid lines to read

(mp+2) grid lines to read

(mp+2) grid lines to read

(mp+2) grid lines to read

grid partitioning & I/O pattern



Check-pointing and solution dump

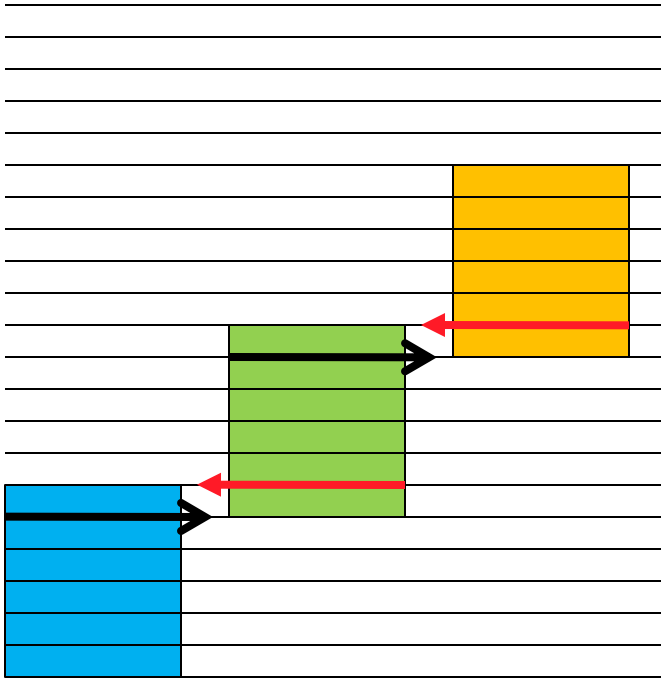
(mp+1) grid lines to write

(mp) grid lines to write

(mp) grid lines to write

(mp+1) grid lines to write

grid partitioning & communication pattern

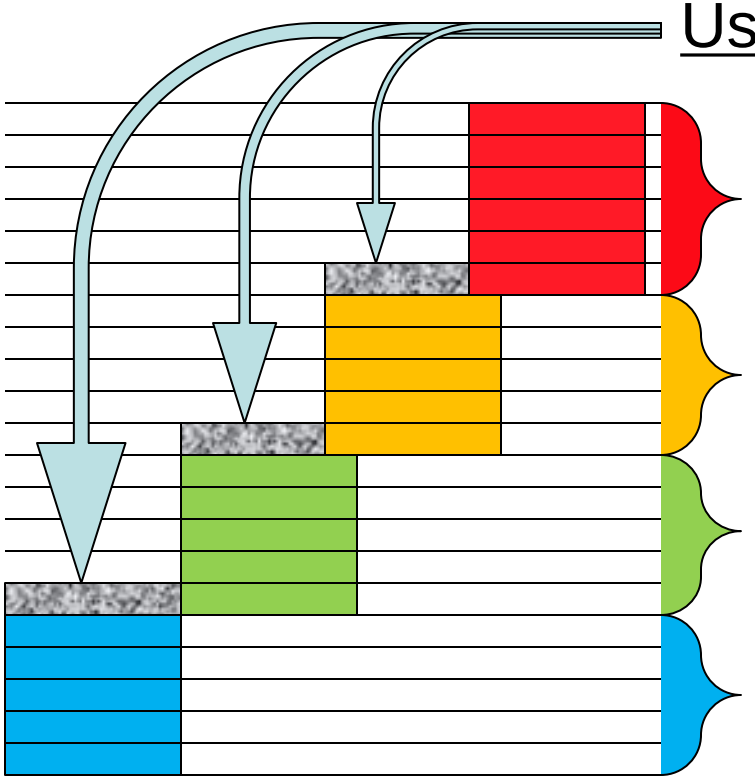


Overlap Send and Receive

Proc. 0 does not receive from “below”

Proc. (N-1) does not send “above”

grid partitioning & *in situ* graphics



Use ghost-nodes to prevent overlapps

(mp+2) lines to send

(mp+1) lines to send

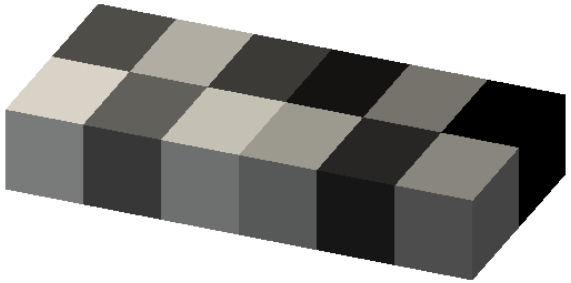
(mp+1) lines to send

(mp+1) lines to send

Visitrectmeshsetrealindices(h, minRealIndex, maxRealIndex)

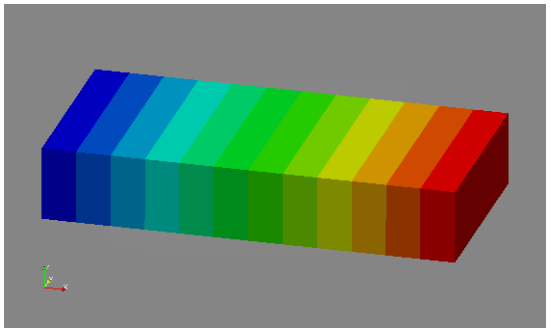
Strategies for load balancing (static)

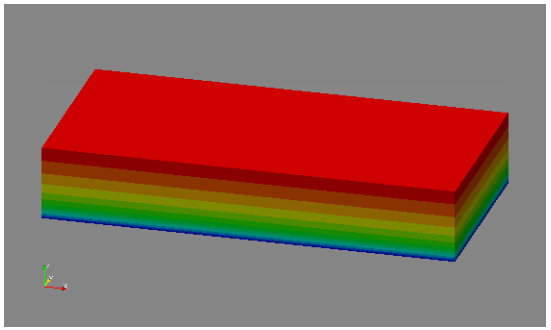
A checkerboard pattern seem like it would give a good compromise, but the communication pattern is more complex to program

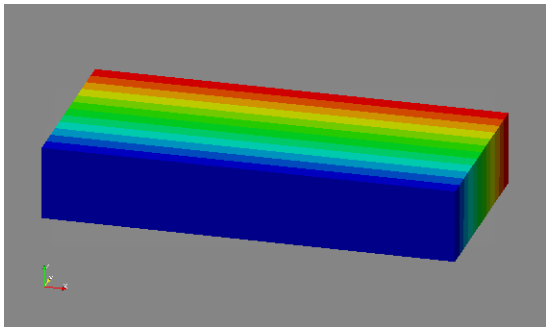


Grid splitting strategies will also affect:

- boundary conditions
- I/O patterns
- *in-situ* ghost regions

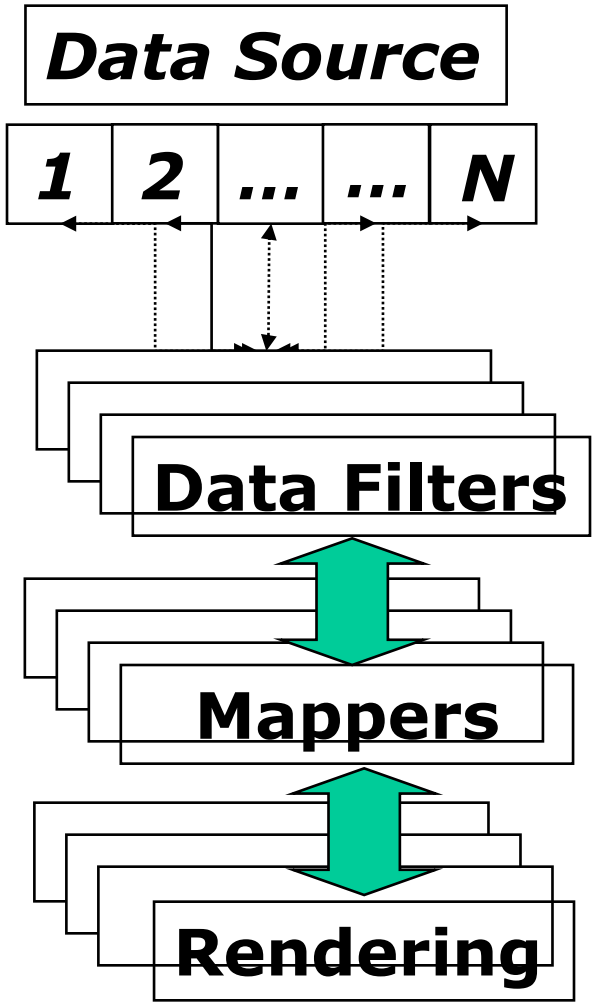




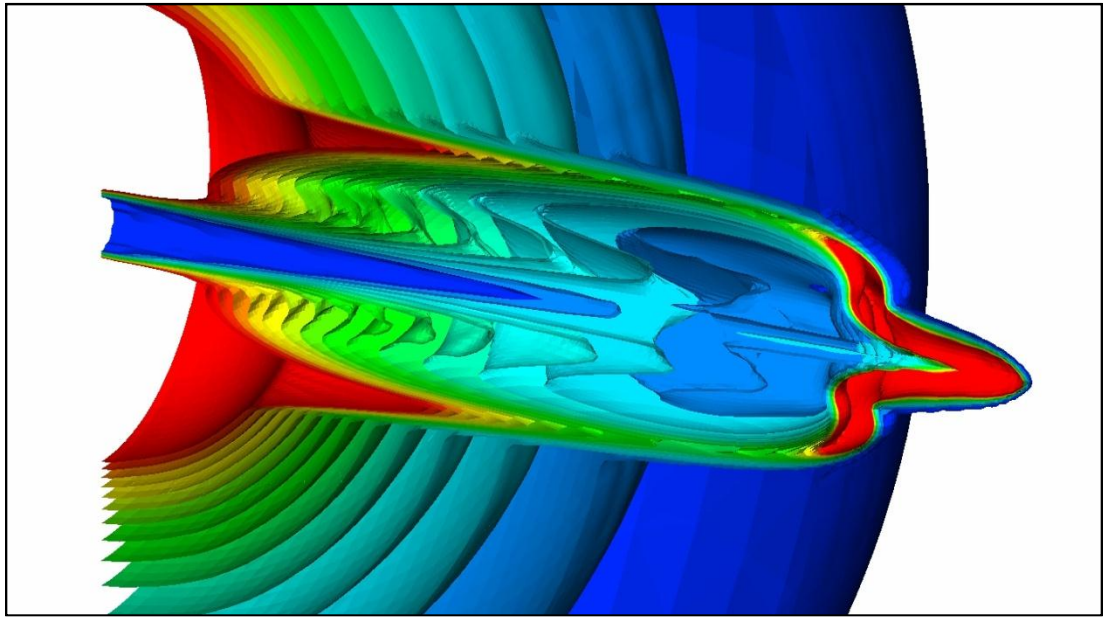


Parallel visualization pipelines

- Data parallelism everywhere
  - Data I/O
  - Processing
  - Rendering
- The Source will distribute data partitions to multiple execution engines



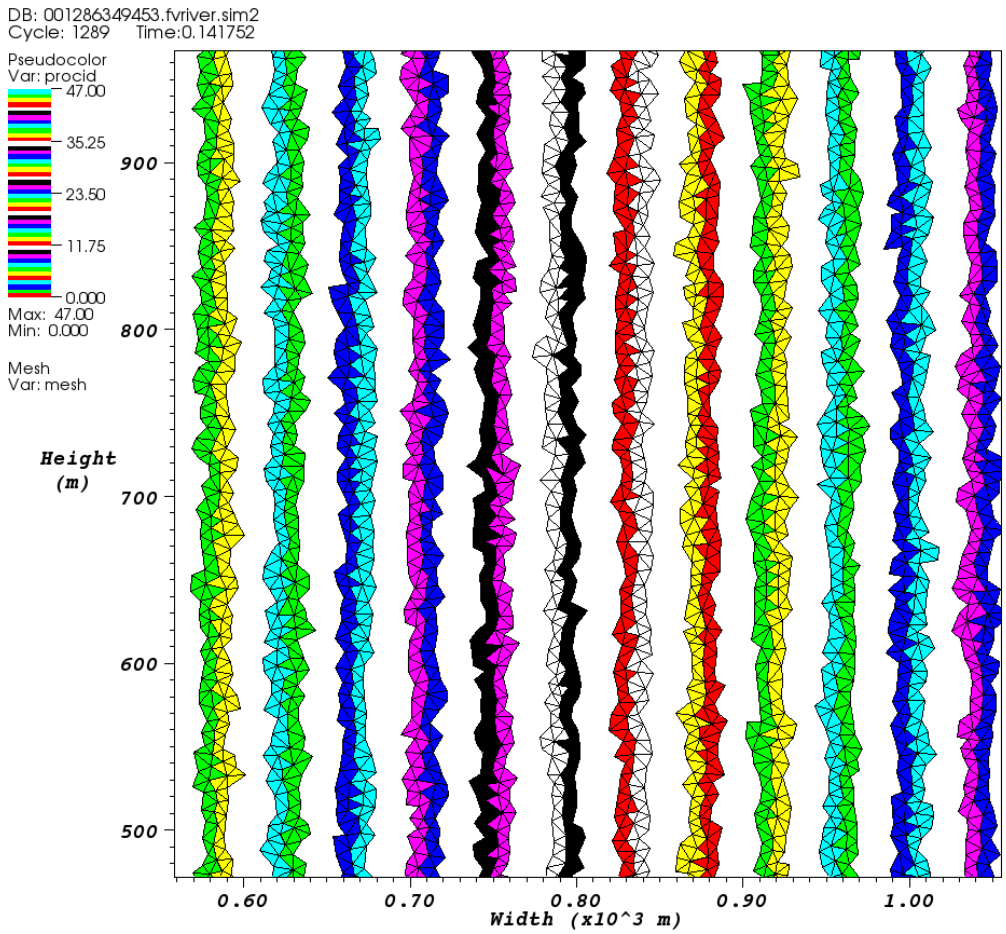
Arbitrary (or adaptive) 3-D data partitioning



Is the final image order-independent?

A sort-last compositing (valid for opaque geometry) enables complete freedom in data partitioning

Example of the use of ghost-cells

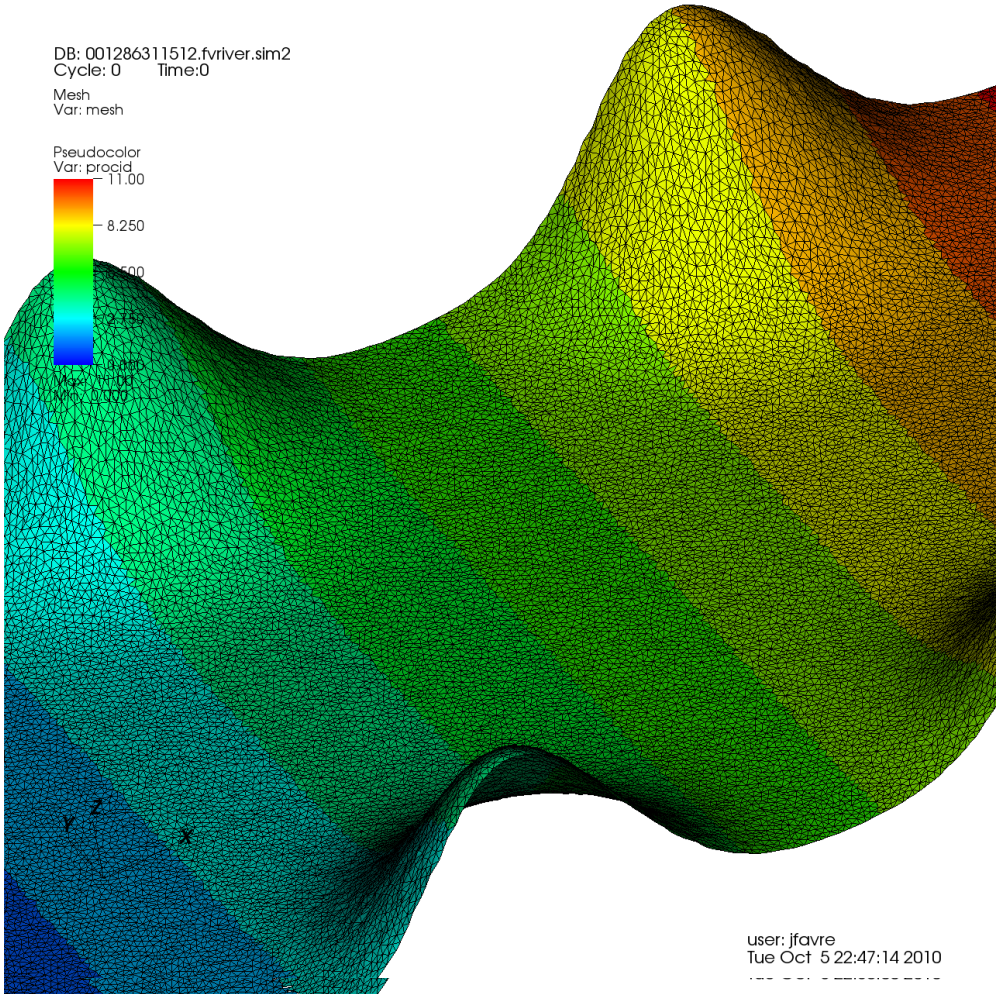


Ghost- or halo-cells are owned by processor P, but reside in the region covered by processor P+k

(k is an offset in the MPI grid)

Their purpose is to guarantee data continuity

Example of the use of ghost-cells



Ghost- or halo-cells are usually not saved in solution files because the overhead can be quite high.

A restart usually involves reading the “normal” grid, and communicate (initialize) ghost-cells before proceeding.

in-situ (parallel) visualization

Could I instrument parallel simulations to communicate to a subsidiary visualization application/driver?

- Eliminate I/O to and from disks
- Use all my grid data with ghost-cells
- Have access to all time steps, all variables
- Use my parallel compute nodes

- Don’t invest into building a GUI, a new visualization package, or a parallel I/O module

Two complementary approaches

- Parallel Data transfer to Distributed Shared Memory
  - Computation and Post-processing physically separated
  - developed at CSCS, publicly available on HPCforge

  or,
  - Use ADIOS (not described here, google “ADIOS NCCS”)

- Co-processing or *in-situ*
  - Computation and Visualization on the same nodes
  - A ParaView project
  - A VisIt project
  - See short article at EPFL
  - Tutorial at PRACE Winter School
  - Tutorial proposed at ISC’11

First approach:

Parallel Data transfer to Distributed Shared Memory

- Live post-processing and steering:
  - Get analysis results whilst the simulation is still running
  - Re-use generated data for further analysis/compute operations

- Main objectives:
  - Minimize modification of existing simulation codes

HDF5 – Virtual File Drivers

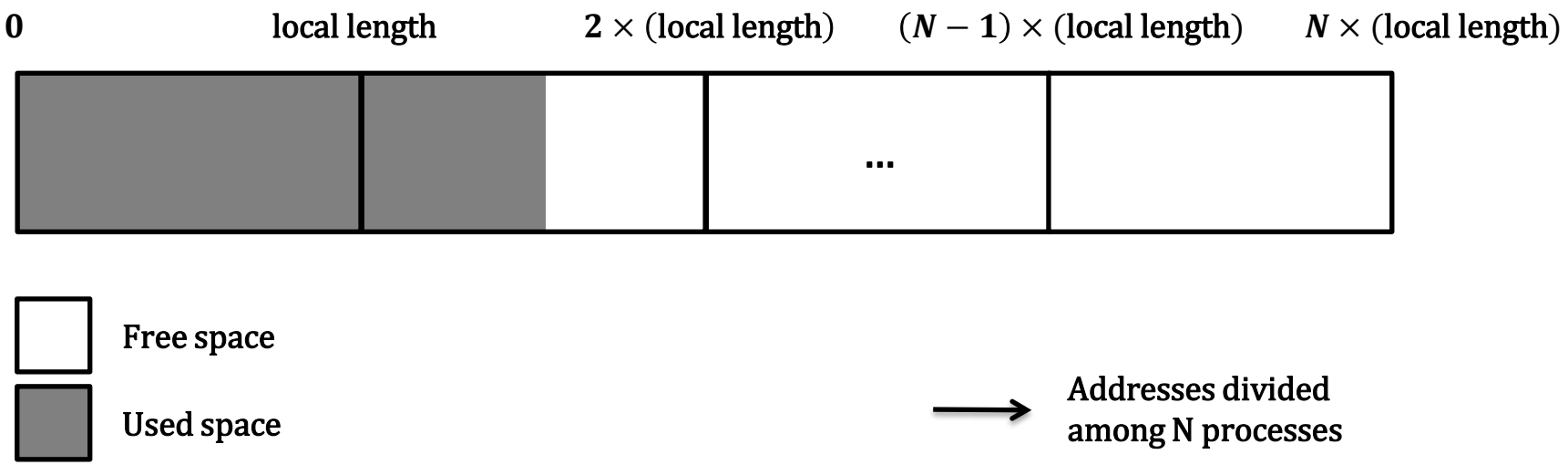
- HDF5 (Hierarchical Data Format):
  - Widely known data format
- HDF5 based on a Virtual File Drivers (VFD) architecture:
  - Modular low-level structure called “drivers” (H5FDxxx)
  - Mapping between HDF5 format address space and storage
- HDF5 already provides users with different file “drivers” (e.g. MPI-IO driver)

HDF5 – Virtual File Drivers

- The CSCS driver: **H5FDdsm**
  - Data sent into HDF5 is then automatically redirected to this driver

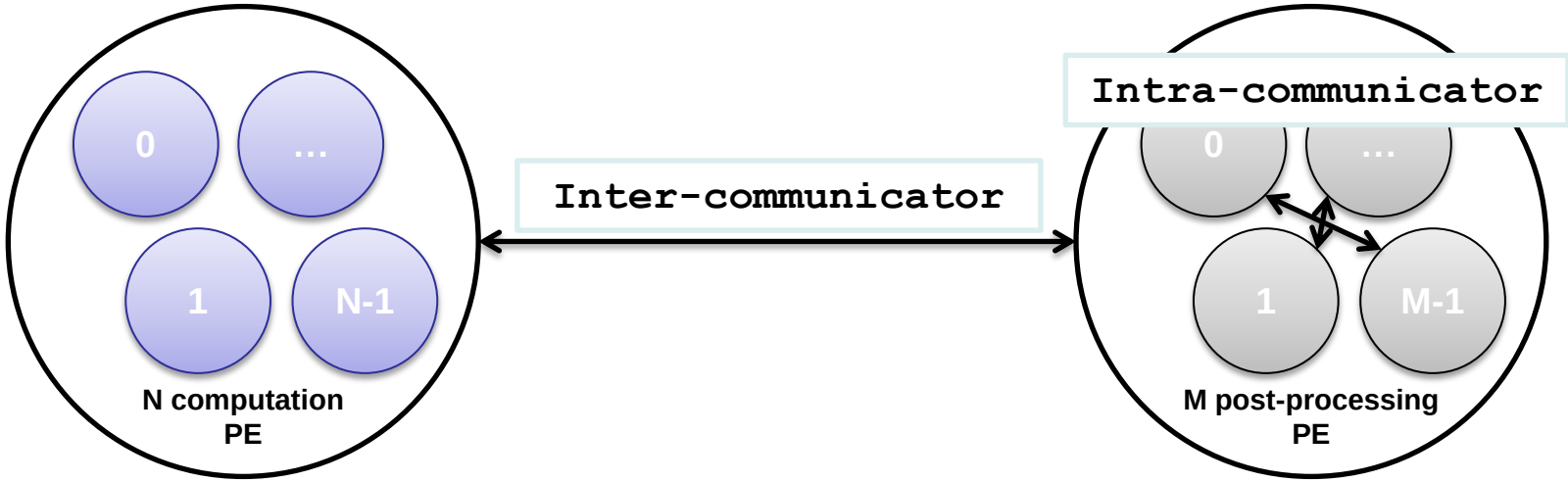
```
H5Pset_fapl_dsm(fapl_id, MPI_COMM_WORLD, NULL);
```

  - Based on the Distributed Shared Memory (DSM) concept



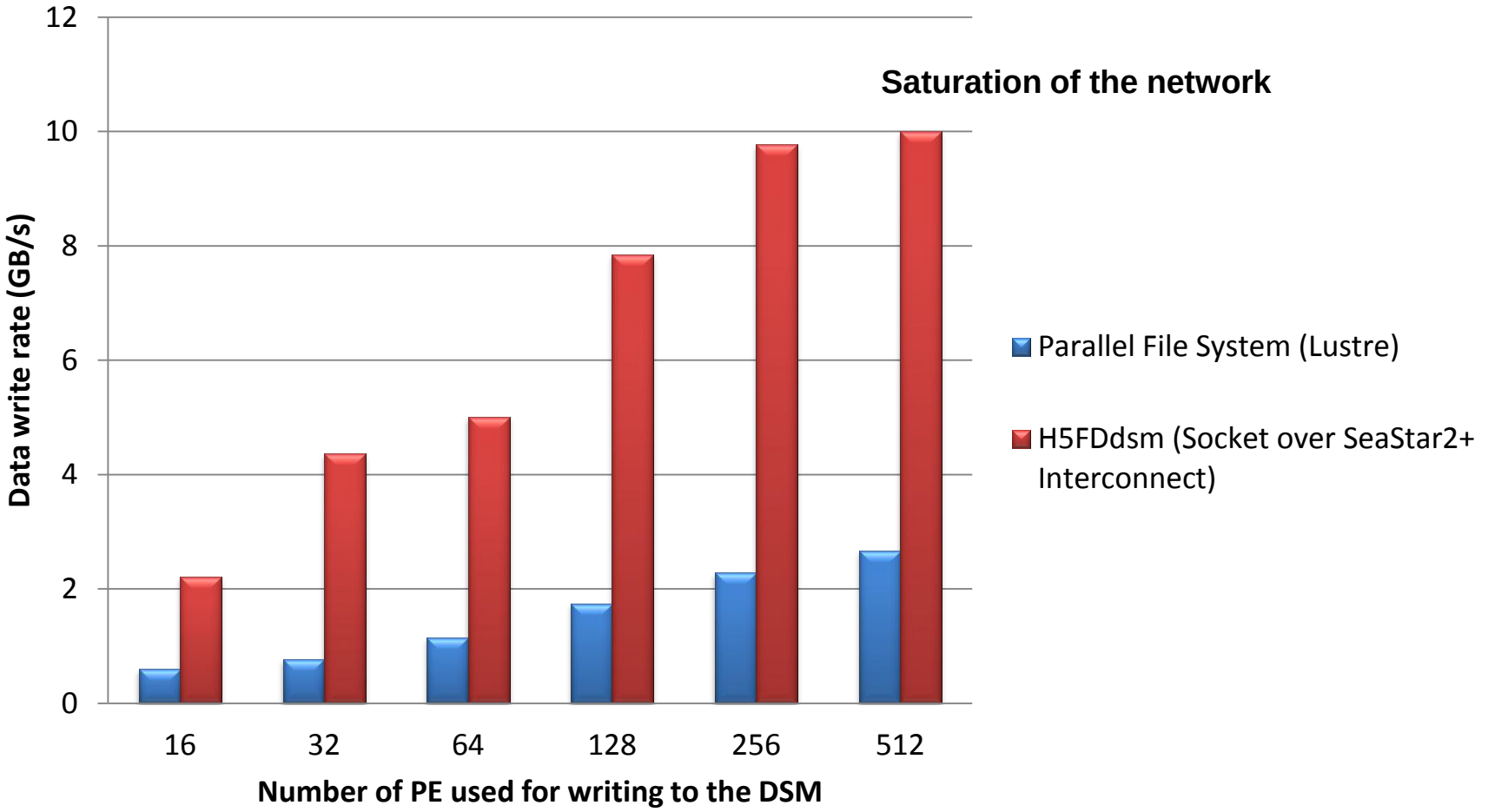
H5FDdsm – Modular communication

- **Inter-communicator**: MPI/socket/(something else) ;
- **Intra-communicator**: always MPI
- Main operating mode:
  - DSM allocated on the post-processing nodes
    - Computation writes using remote put



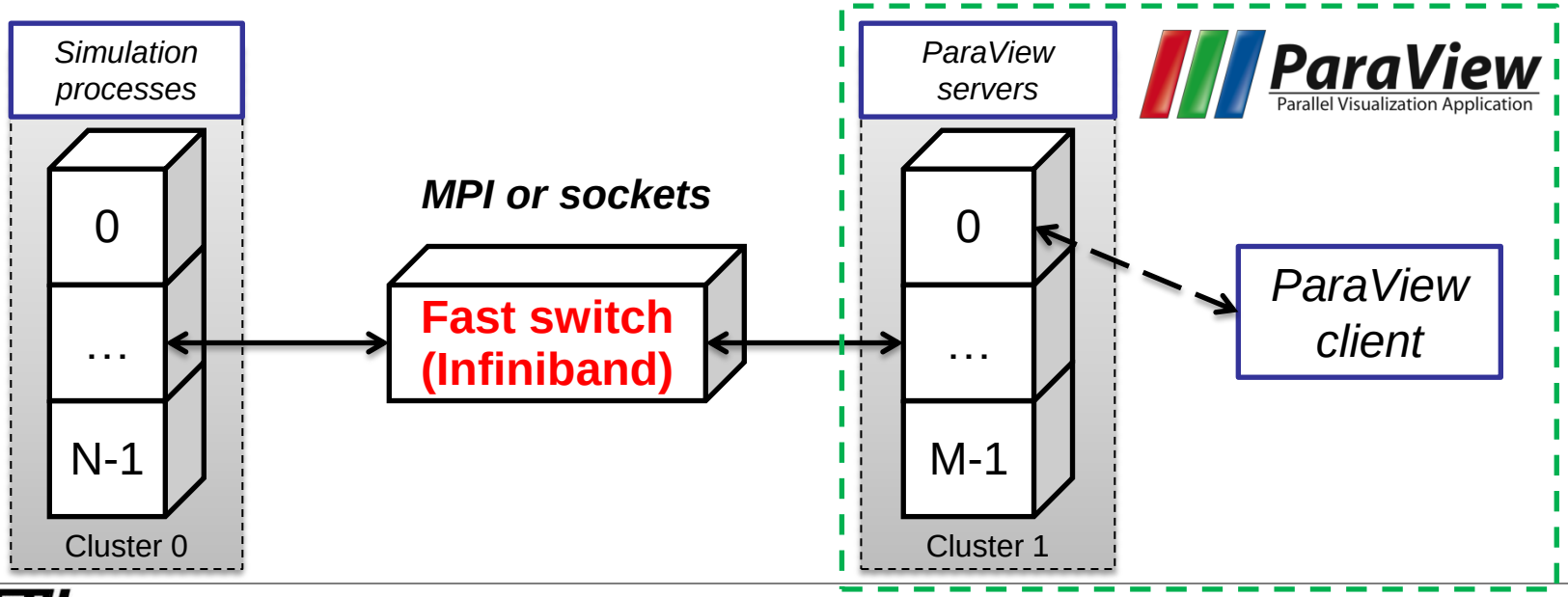
Write test with a 20GB DSM

distributed among 8 post-processing nodes the XT5



The DSM interface within ParaView

- Pause, Play, Switch to disk implemented within the driver
  - No additional calls necessary
- Select datasets to be sent on the fly
- Restart simulation from HDF file stored in the DSM
- Modify parameters/arrays/etc – share datasets back to the simulation



Advantages

- Parallel real-time post-processing of the simulation
- Easy to use if application uses HDF5 already
- Do not need to have a big amount of memory on compute nodes since everything is sent to a remote DSM
- Not all post-processing apps scale as well as the simulation
  - Run PV on fat nodes and simulation on a very large number of nodes

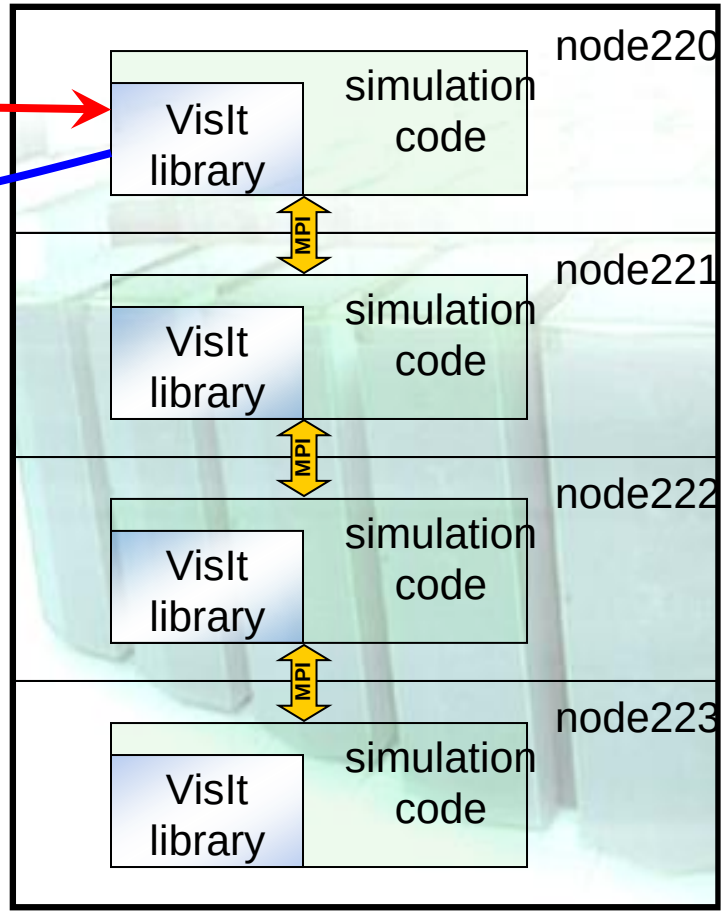
Second Method: VisIt

Desktop Machine

commands

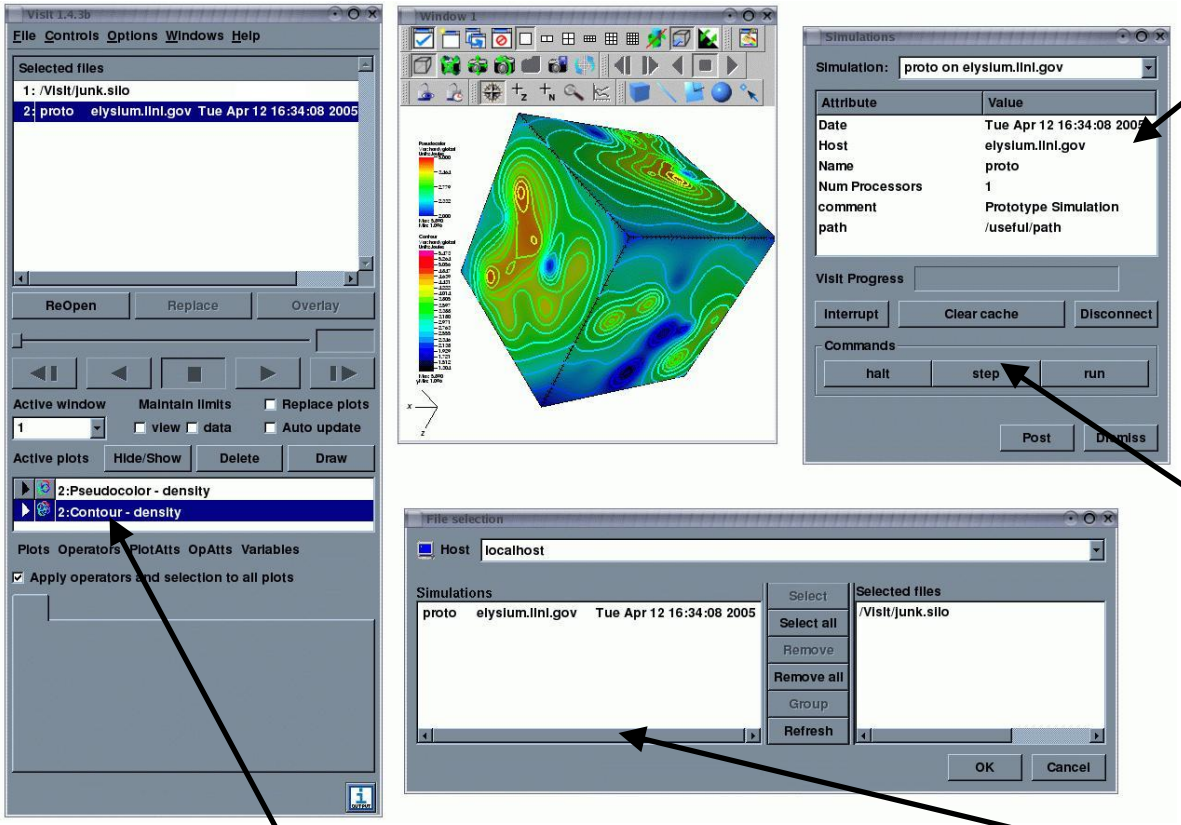
images

Parallel Supercomputer



Link simulation with visualization library and drive it from a GUI

Use VisIt https://wci.llnl.gov/codes/visit



The Simulation's window shows meta-data about the running code

Control commands exposed by the code are available here

Users select simulations to open as if they were files

All of VisIt's existing functionality is accessible

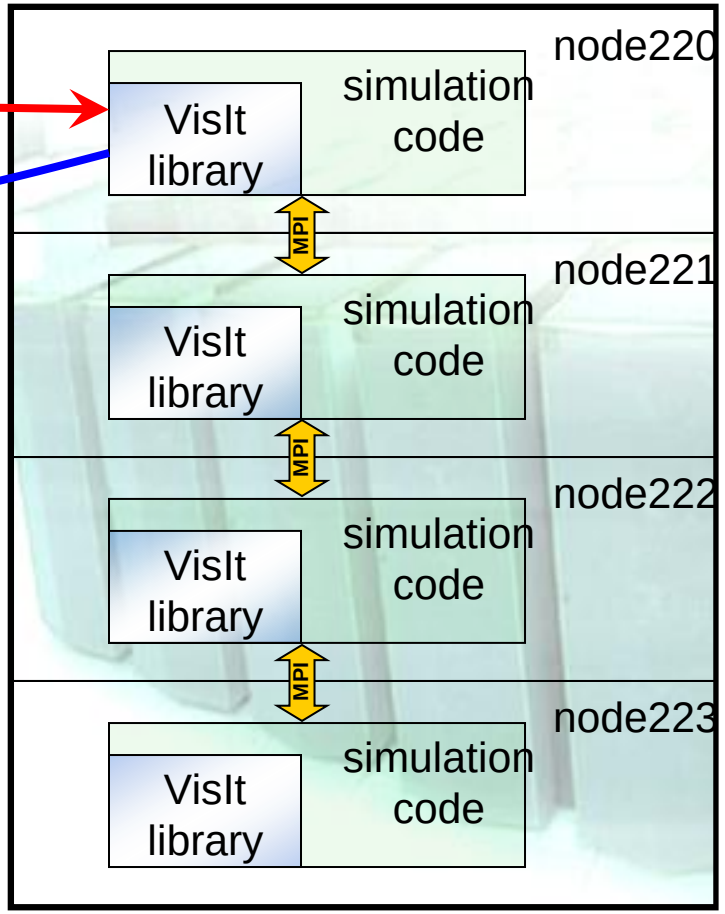
VisIt's plotting is called from the simulation

Desktop Machine

commands

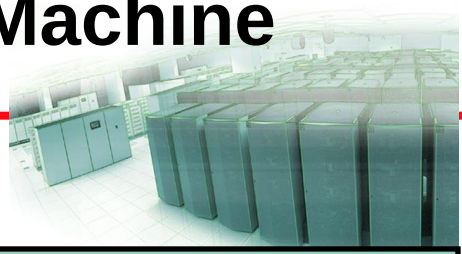
images

Parallel Supercomputer



No pre-defined visualization scenario needs to be defined

Launch Simulation on Big Parallel Machine



Linux desktop machine

BPM login node

PSUB/SRUN

BPM Home Directory

SimCode0 data — bpm20

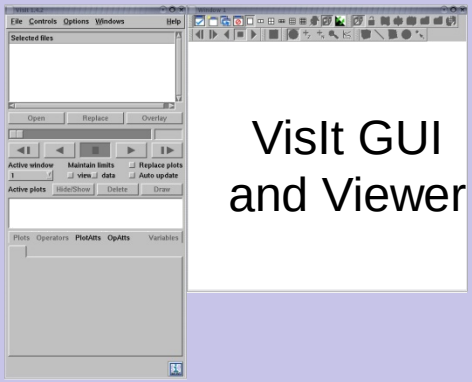
SimCode1 data — bpm21

SimCode2 data — bpm22

SimCode3 data — bpm23

Remote VisIt task connects to Simulation

Linux desktop machine



VisIt GUI and Viewer

BPM login node

VisIt Launcher

BPM Home Directory

~/.visit/simulations

jet00 bpm33 6666 May1

jet01 bpm20 2345 May1

SimCode0 listening data — bpm20

SimCode1 data — bpm21

SimCode2 data — bpm22

SimCode3 data — bpm23

Simulation *becomes* engine, connects to Viewer

CSCS
Swiss National Supercomputing Centre

VisIt Launcher

BPM login node

Linux desktop machine

VisIt GUI and Viewer

BPM Home Directory

~/.visit/simulations

jet00 bpm33 6666 May1
jet01 bpm20 2345 May1

SimCode0 listening
VisIt Engine
data
bpm20

SimCode1
VisIt Engine
data
bpm21

SimCode2
VisIt Engine
data
bpm22

SimCode3
VisIt Engine
data
bpm23

VisIt requests pull Data from Simulation

BPM login node

VisIt Launcher

Linux desktop machine

VisIt GUI and Viewer

BPM Home Directory

~/.visit/simulations

jet00 bpm33 6666 May1
jet01 bpm20 2345 May1

bpm20

SimCode0 listening
VisIt Engine
data interface
data

bpm21

SimCode1
VisIt Engine
data interface
data

bpm22

SimCode2
VisIt Engine
data interface
data

bpm23

SimCode3
VisIt Engine
data interface
data

Some details on the APIs

- The C and Fortran interfaces for using SimV2 are identical, apart from calling different function names

- The VisIt Simulation API has just a few functions
  - set up the environment
  - open a socket and start listening
  - process a VisIt command
  - set the control callback routines
- The VisIt Data API has just a few callbacks
  - GetMetaData()
  - GetMesh()
  - GetScalar(), etc

Main program

```
int main(int argc, char **argv)
{
    SimulationArguments(argc, argv);


    read_input_deck();

    mainloop();
    return 0;
}
```

Main program is augmented

```
int main(int argc, char **argv)
{
    SimulationArguments(argc, argv);
    VisItSetupEnvironment();
    VisItInitializeSocketAndDumpSimFile();
    read_input_deck();

    mainloop();
    return 0;
}
```

Example with our Shallow Water Simulation code:

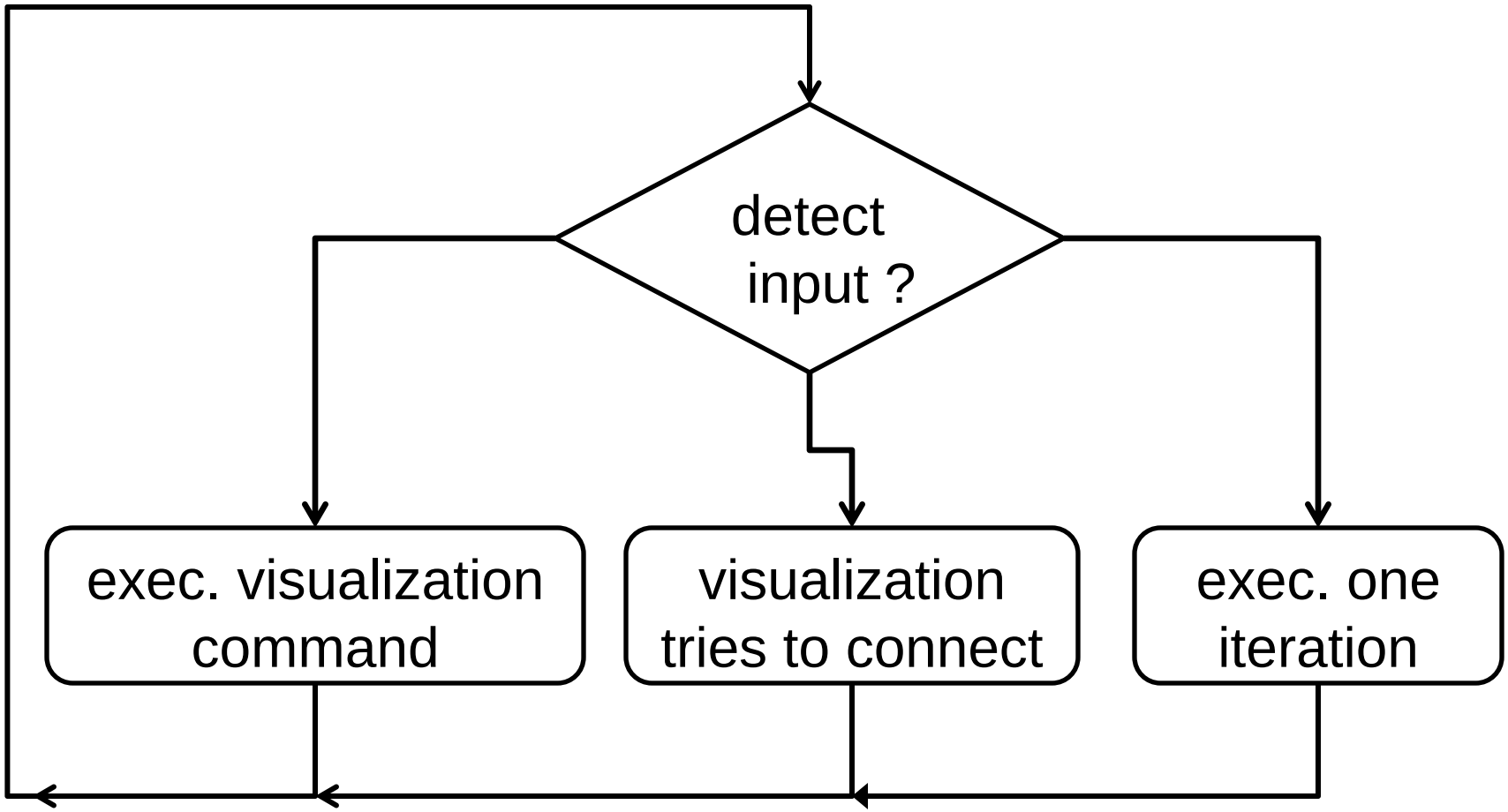
Additions to driver program:

15 lines of F95 code

mainloop() will be augmented

```
void mainloop(simulation_data *sim)
{
    int blocking, visitstate, err = 0;
    do
    {

    simulate_one_timestep(sim);

    } while(!sim->done && err == 0);
}
```

Flow diagram



Simulation main loop is augmented

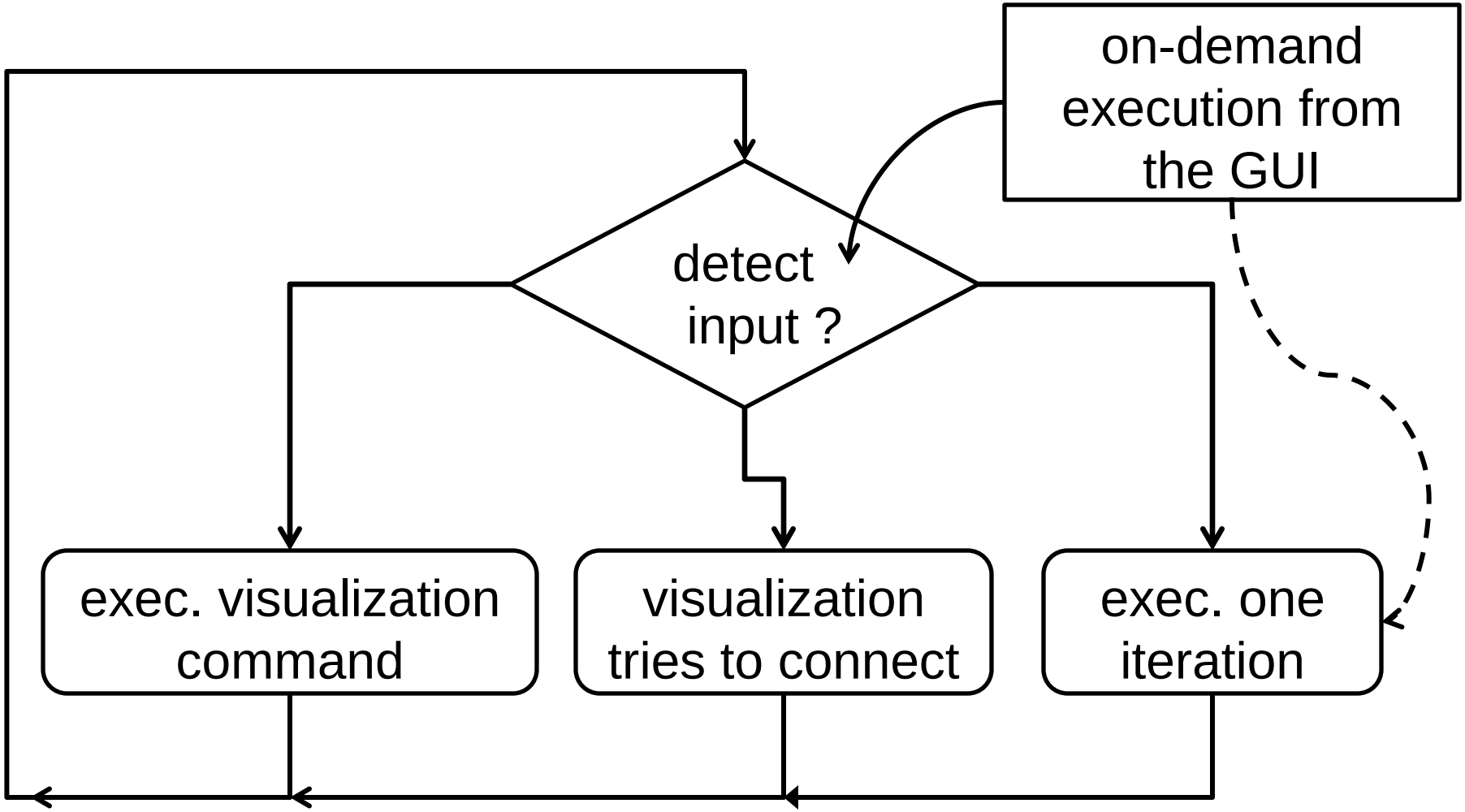
Example with our Shallow Water Simulation code:

Additions to simulation loop:

53 lines of F95 code

```
void mainloop(simulation_data *sim)
{
    int blocking, visitstate, err = 0;
    do
    {
    blocking = (sim->runMode == SIM_RUNNING) ? 0 : 1;
    visitstate = VisItDetectInput (blocking, -1);

    if(visitstate == 0) {
       simulate_one_timestep(sim);
       }
```

Two entry points to the execution



Callbacks are added to advertize the data

visitcommandcallback()

visitgetmetadata ()

- Mesh name
- Mesh type
- Topological and spatial dimensions
- Units, labels
- Variable names and location (cell-based, node-based)
- Variable size (scalar, vector, tensor)
- Commands which will be understood (next(), halt(), run(), …)

visitsimgetmesh()

Example with our Shallow Water Simulation code:

New file to link with the rest of simulation code:

432 lines of F77 code

Get_Mesh() & GetVariable() must be created

integer function visitgetmesh(domain, name, lname)

parameter (NX = 512)
parameter (NY = 512)
real rmx(NX), rmy(NY)

allocate data handles for the coordinates and advertize them

F90 example:

visitrectmeshsetcoords(h, x, y)

How much impact in the source code?

The best suited simulations are those allocating large (contiguous) memory arrays to store mesh connectivity, and variables

Memory pointers are used, and the simulation (or the visualization) can be assigned the responsibility to de-allocate the memory when done.

F90 example:

allocate (v(0:m+1,0:mp+1))

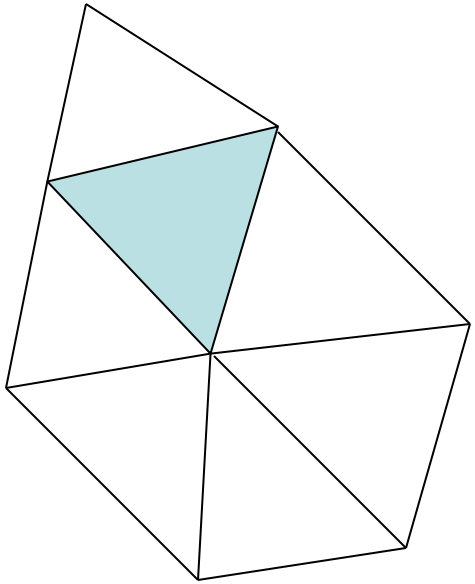
visitvardatasetd(h, VISIT_OWNER_SIM, 1, (m+2)(mp+2), v)*

How much impact in the source code?

The least suited are those pushing the Object Oriented philosophy to a maximum.

Example: Finite Element code handling a triangular mesh:

```
TYPE Element
        REAL(r8) :: x(3)
        REAL(r8) :: y(3)

        REAL(r8) :: h
        REAL(r8) :: u
        REAL(r8) :: zb(3)

END TYPE Element
```



How much impact in the source code?

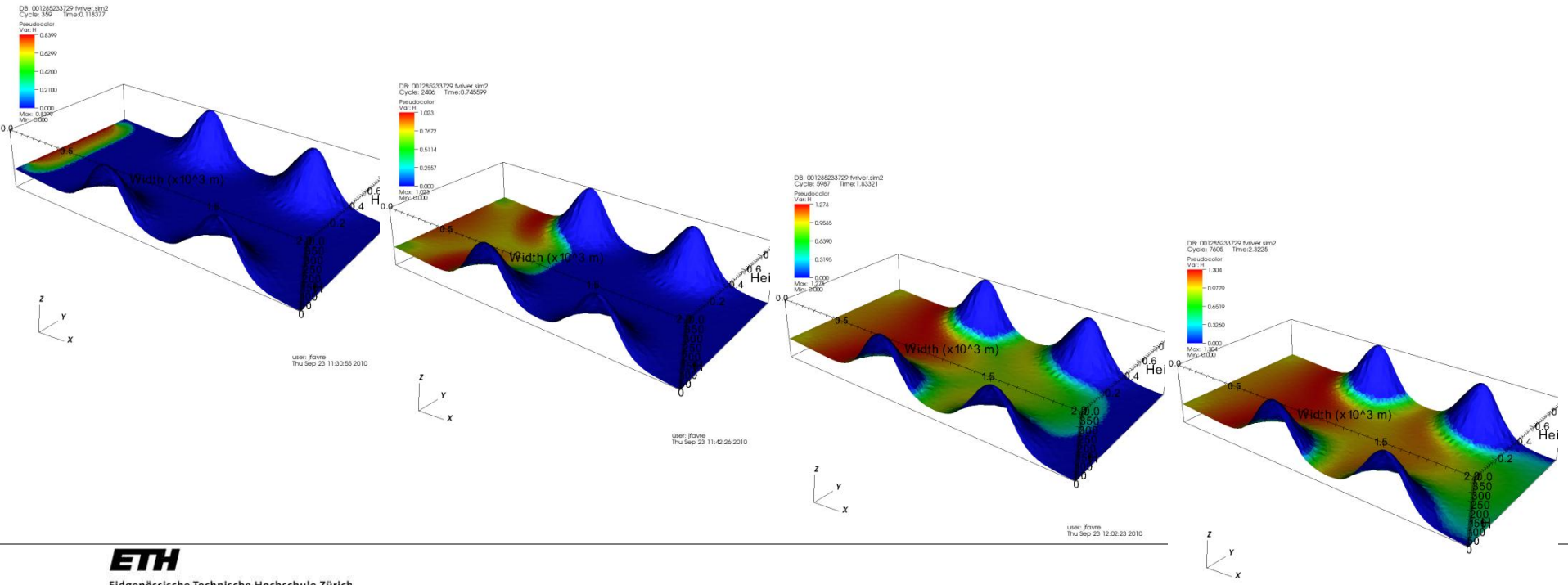
When data points are spread across many objects, there must be a new memory allocation and a gathering done before passing the data to the Vis Engine

```
REAL, DIMENSION(:), ALLOCATABLE :: cx

ALLOCATE( cx(numNodes) , stat=ierr)

DO iElem = 1, numElems+numHalos
      DO i = 1, 3
        cx(ElementList(iElem)%lclNodeIDs(i)) = ElementList(iElem)%x(i)
      END DO
END DO

err = visitvardatasetf(x, VISIT_OWNER_COPY, 1, numNodes, cx)
```

VisIt can control the running simulation

- Connect and disconnect at any time while the simulation is running
- We program some buttons to react to the user's input:

"Halt", "Step", "Run", "Update", "others…"



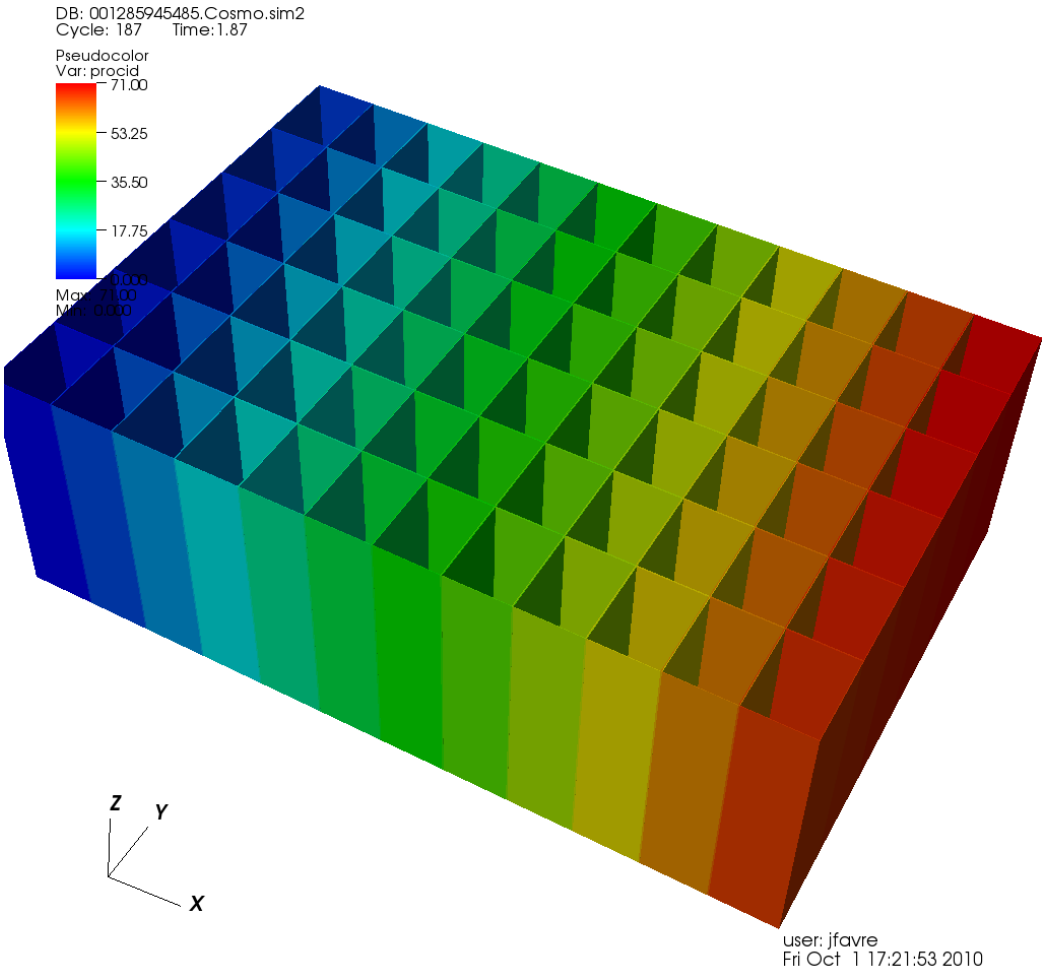
Ex: Domain distributed over MPI tasks

Dimensions = 1080x720x360
10 variable arrays (double)
MPI Partition = 12x6

Data per MPI task: 360 Mbytes

Application linked with (resp. without) the libsim library = 977 Kb (resp. 850 Kb)

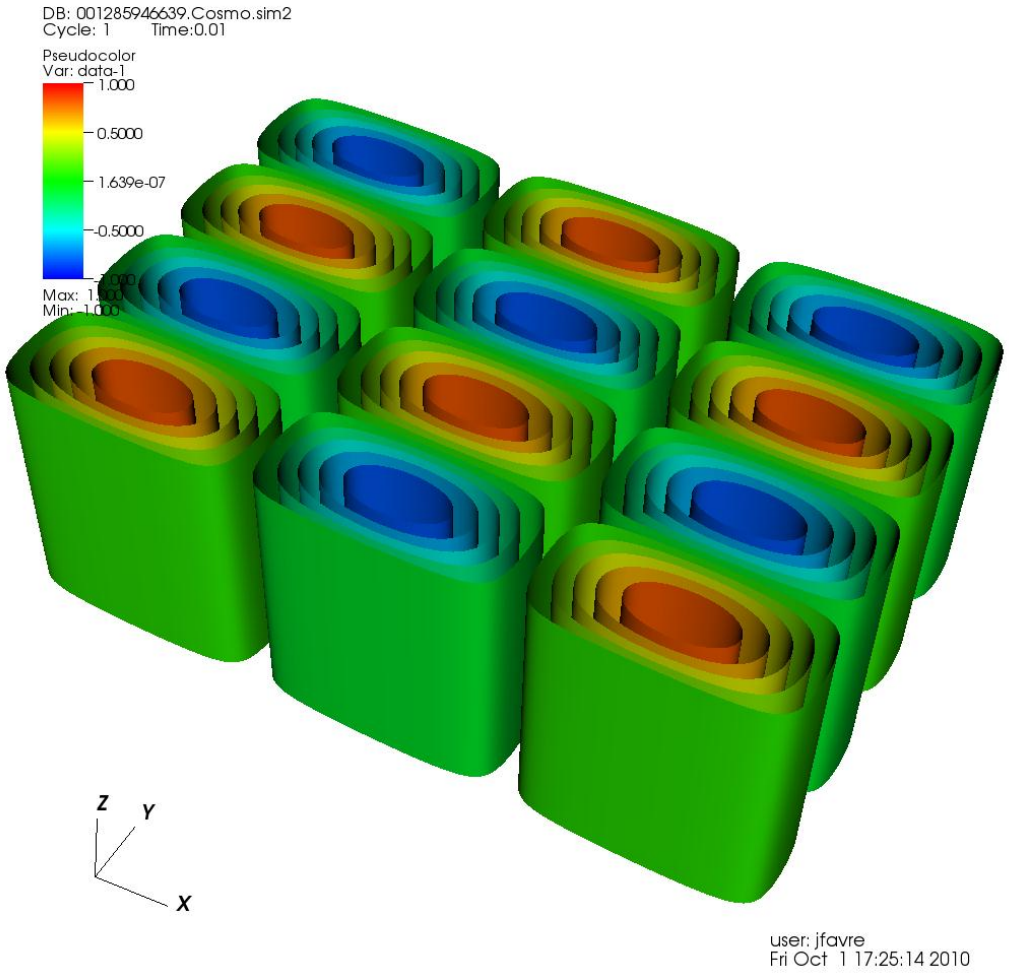
At runtime, one extra lib is loaded (306 Kb)



Domain distributed over multiple MPI tasks

If graphics is very heavy, it is done remotely (by the simulation), and sent over to the client as a pixmap.

If light, geometry is sent to client for local rendering



The *in-situ* library provides many features

- Access to scalar, vector, tensor arrays, and label
- CSG meshes
- AMR meshes
- Polyhedra
- Material species
- Ability to save images directly from the simulation
- Interleaved XY, XYZ coordinate arrays

Advantages compared to saving files

- The greatest bottleneck (disk I/O) is eliminated
- Not restricted by limitations of any file format
- No need to reconstruct ghost-cells from archived data
- All time steps are potentially accessible
- All problem variables can be visualized
- Internal data arrays can be exposed or used
- Step-by-step execution will help you debug your code and your communication patterns

- The simulation can watch for a particular event and trigger the update of the VisIt plots

In the past, we focused on raw data => images

<DataArray type="UInt8" Name="types" format="appended" RangeMin="" RangeMax="" offset="5948" />

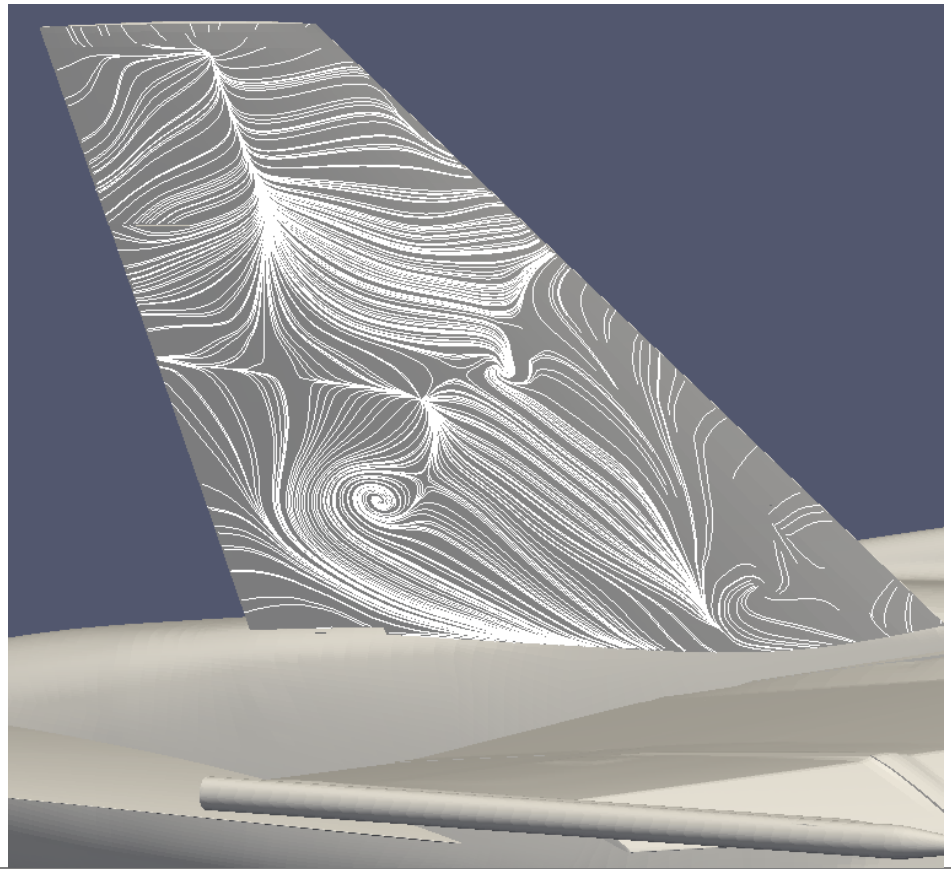
</Cells>

</Piece>

</UnstructuredGrid>

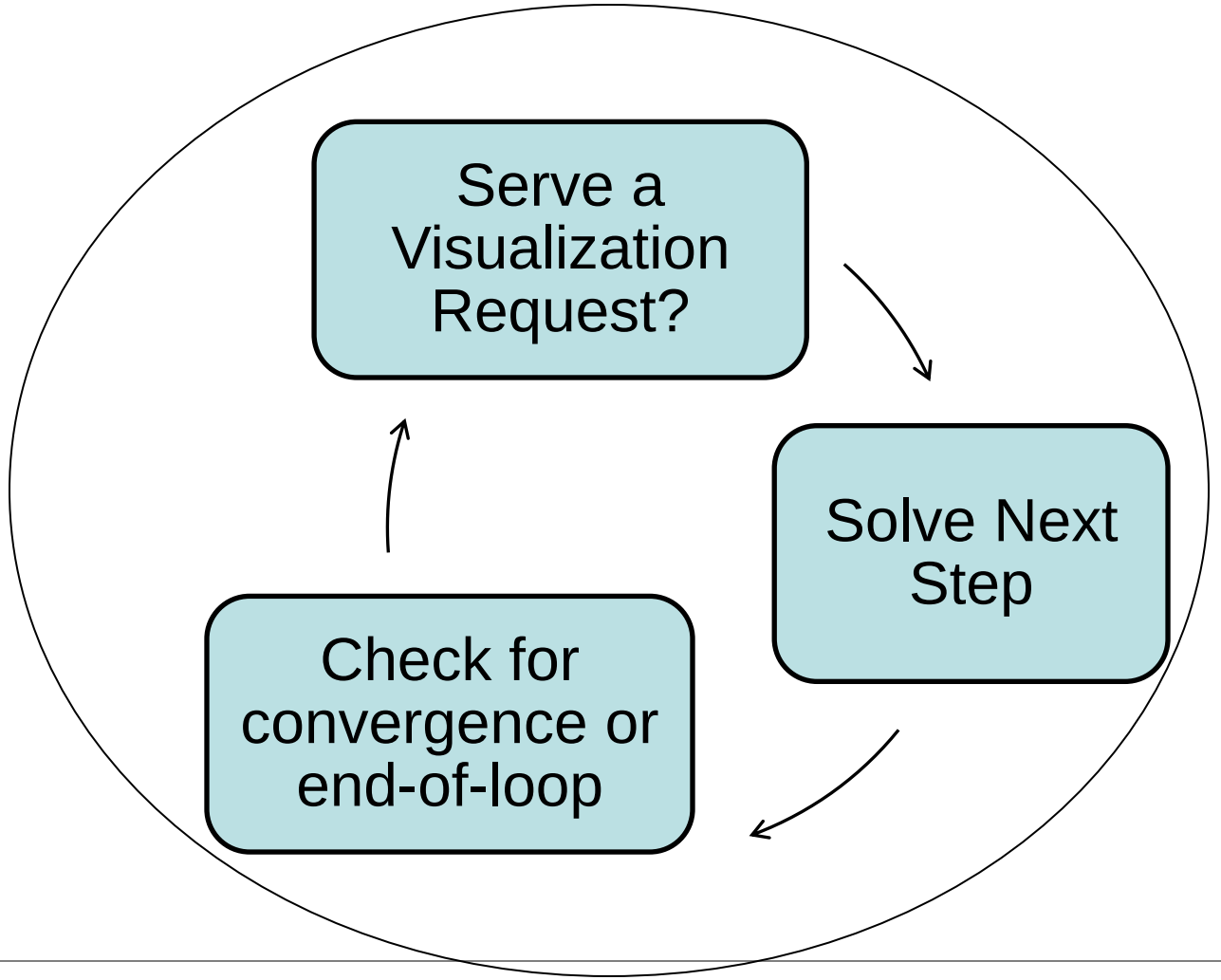
<AppendedData encoding="base64">

_AQAAPAAAAFwAAAA==eJwVzzEoRHE
Ax/H/YDAYbjAYDDcYDIYbDAbluQwGww
0Gg+EGg8Fwg8FgeEm6JF2SLkkvSZekS
9J1SS9Jl6RL0iXpjUaj0Uf9PvOvbwgfxRA+
+SIjngohREZMQkpGmA4hR54CESXKVIip
UqNOQoMmLVI6dOmR8c0PvwT/ffQzQI5
BhhgmzwijjFFgnAkmiZhhljlKzLPAImWWW
GaFCqussf7fzgabbFFlmx12qbHHPgfUOe
SIYxJOOOWMBudccEmTK665oUWbW+5
lueeBRzo88cwLXV55451e8Q8G5lcqAQA
AAACAAABABQAAgQIAAA

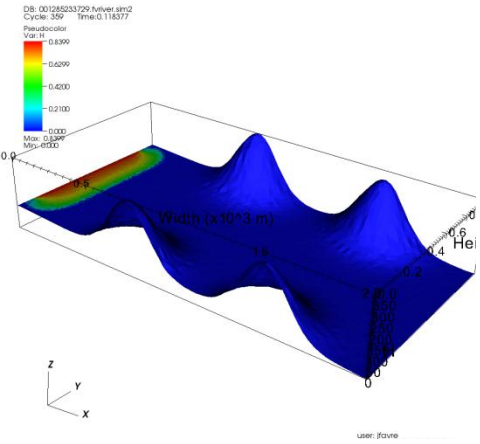


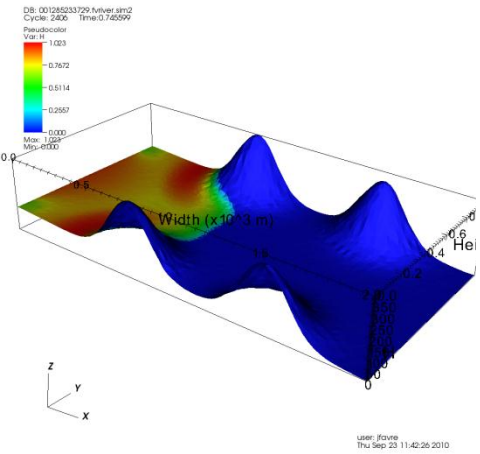
We are now adding a new interaction paradigm

mega-,
giga-,
peta-,
exa-scale
simulations
can now be
coupled with
visualization

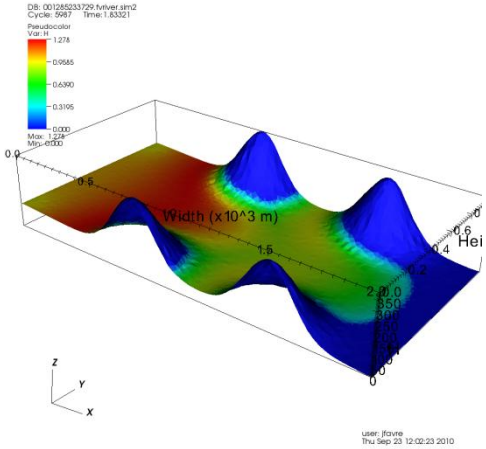


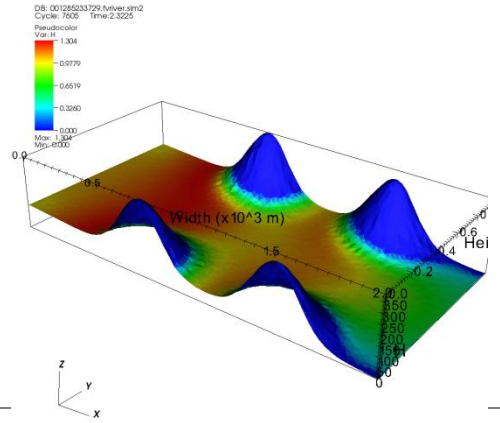
now focus on source code => live images






```
REAL, DIMENSION(:), ALLOCATABLE :: cx
ALLOCATE( cx(numNodes) , stat=ierr)

DO iElem = 1, numElems+numHalos
      DO i = 1, 3
        cx(ElementList(iElem)%lclNodeIDs(i))
= ElementList(iElem)%x(i)
      END DO
END DO

err = visitvardatasetf(x,
VISIT_OWNER_COPY, 1, numNodes, cx)
```





We need a new data analysis infrastrcuture

- Domain decomposition optimized for simulation is often unsuitable for parallel visualization
- To optimize memory usage, we must share the same data structures between simulation code and visualization code to avoid data replication
- Create a new vis infrastructure, develop in-situ data encoding algorithms, indexing methods, incremental 4D feature extraction and tracking
- Petascale visualization tools may soon need to exploit new parallel paradigms in hardware, such as multiple cores, multiple GPUs, cell processors…

Conclusion

Parallel visualization is a mature technology, but was optimized as a stand-alone process. It can run like a supercomputer simulation, but is also limited by I/O.

In-situ visualization is an attractive strategy to mitigate this problem, but will require an even stronger collaboration between the application scientists and the visualization scientist, and the development of a new family of visualization algorithms

Demonstrations