Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver

John Biddiscombe[1], Jerome Soumagne[1,3], Guillaume Oger[2], David Guibert[4], Jean-Guillaume Piccinali[1]

[1]Swiss National Supercomputing Centre, 6928 Manno, Switzerland
[2]HydrOcean, 44321 Nantes, France
[3]INRIA Bordeaux Sud-Ouest, 33405 Talence, France
[4]Ecole Centrale Nantes, France

Abstract

We present a framework for interfacing an arbitrary HPC simulation code with an interactive ParaView session using the HDF5 parallel IO library as the API. The implementation allows a flexible combination of parallel simulation, concurrent parallel analysis and GUI client, all of which may be on the same or separate machines. Data transfer between the simulation and the ParaView server takes place using a virtual file driver for HDF5 that bypasses the disk entirely and instead communicates directly between the coupled applications in parallel. The simulation and ParaView tasks run as separate MPI jobs and may therefore use different core counts and/or hardware configurations/platforms, making it possible to carefully tailor the amount of resources dedicated to each part of the workload. The coupled applications write and read 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 reread by the simulation (or vice versa). This allows not only simple parameter changes, but complete remeshing of grids, or operations involving regeneration of field values over the entire domain, to be carried out. To avoid the problem of manually customizing the GUI for each application that is to be steered, we make use of XML templates that describe outputs from the simulation, inputs back to it, and what user interactions are permitted on the controlled elements. This XML is used to generate GUI and 3D controls for manipulation of the simulation without requiring explicit knowledge of the underlying model.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.1]: Parallel Processing—Computer Graphics [I.3.2]: Distributed/network graphics—Software Engineering [D.2.2]: Software libraries—

1. Introduction

Scientists and engineers scaling-up their codes to run on HPC platforms may still wish to experiment with algorithms and optimizations that affect the performance or the accuracy of the results they obtain. Often, it is desirable to play with variables during a run to see how they affect a particular parameter (or derived result). When these parameters are simple scalar values that control the algorithm it is possible to write them to a file periodically and allow the simulation to pick them up on the fly – a procedure that has been used since the birth of scientific computing. When the parameter to be controlled is more complex, or is used to produce further data that is less obviously understood, it may be desirable to have a user interface, which allows the interactive adjustment of parameters with immediate feedback on the effects they have. The driving force behind the work presented in this paper has been on the modeling of fluid flows – and in particular fluid structure interactions, using particle methods and boundary geometries. The interactions of the particles and the geometries can produce deformations and motions that are to be studied, and the placement of geometries can dramatically affect the results of individual simulations. For this reason an interface that allows the user to interact with geometries, perform translations, rotations and even re-meshing operations whilst the simulation continues, is required. An additional consideration is that not one, but

DOI: 10.2312/EGPGV/EGPGV11/091-100

four codes (from different teams within the project, using different programming languages and different data models) are candidates for the steering environment and it should be capable of interfacing with them all.

Building a GUI to control and visualize an individual simulation can be a time consuming task, particularly when the simulation is running on a cluster or supercomputer and not on the developers workstation. For this reason, a solution involving an existing application – in this case ParaView [Hen05] – capable of parallel analysis was sought. The developments made to achieve this goal are described as follows; in section 2 we discuss the IO coupling of simulation and steering along with modes of operation and synchronization issues. Section 3 outlines the development of a plugin for ParaView and how the simulation description is created and interpreted to produce a controlling environment. Section 4 deals with a specific application example where an SPH code has been coupled and controlled by ParaView. Finally we compare our implementation to other solutions and draw conclusions.

2. DSM Interface

In [SBC10], an approach was presented to in-situ post-processing using a *Distributed Shared Memory* (DSM) as the interface between arbitrary simulation and post-processing applications using the HDF5 API for the exchange of data. HDF5 supports virtual file driver (VFD) extensions that allow the customization of IO so that the standard disk writer may be replaced by an alternative mechanism. When an application makes use of the DSM VFD, the HDF5 library transparently re-routes all the data transfers to a distributed shared memory buffer allocated on a set of remote nodes reachable via the network. The simulation writes data (in parallel) to this virtual *file*, the controlling environment reads (in parallel) from this file and performs additional/post-processing operations as desired. The original design presented in [SBC10] catered only for write operations by the simulation followed by reads from the host application, but this has now been extended (see [SB11]) to support bi-directional read/write access using a file lock to restrict access from one side whilst the other is using it.

2.1. Synchronization

The DSM 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 driver itself may use different modules for communication between processes, one based on sockets and one based on the MPI layer, which can also take advantage of the RMA implementation provided by MPI (when supported) if requested. For communication within or between processes the terms of intra-communicator and inter-communicator are used:

1. An **intra-communicator** represents the communicator used for internal communications by a given application or job, this communicator always uses the MPI interface;
2. An **inter-communicator** links two different applications or two different sets of processes together and uses either an MPI or a socket interface to connect them.

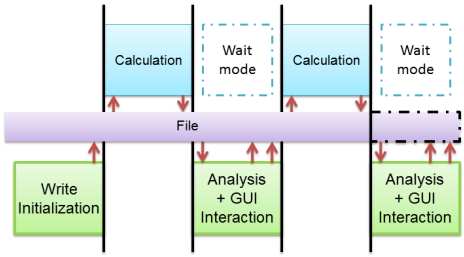
The client task is unchanged by the use of the DSM driver (as compared to a standard MPI-IO driver), but the server task requires an additional thread that is responsible for handling communication requests served by the **inter-communicator**. When using socket or standard MPI, a handshaking process takes place for each block of write operations, but when RMA is available in the MPI distribution, writes from the client may take place without this extra message overhead. In addition, when RMA is used, the DSM synchronization mechanism (which was required when making multiple send/receives during the create and close operations to maintain file state coherency), is simplified by using an MPI window fence synchronization. Memory buffers on the client side are reduced as data is written directly into the server side memory. For these reasons, the RMA method is the preferred protocol. The latest specialized architectures developed for HPC increasingly make use of MPI implementations that support RMA operations at the hardware level and give excellent performance [GT07] in terms of raw throughput of data.

Any operation, which modifies metadata in the file, flags the driver that a synchronization step must take place prior to subsequent accesses – and it is this metadata synchronization that dictates that only one side of the connection may make use of the file at any time. Parallel IO within HDF5 requires collective operations between all nodes using the same file when metadata changes are made: providing the client – or the server – use the file independently, the HDF5 layer handles local synchronization, but if both sides were to attempt to (read/)write concurrently, an additional exchange of metadata would be required between tasks that we do not yet support (currently the metadata is flushed to the file by the HDF5 interface and becomes available automatically when the file is closed and control is handed over). We therefore operate using a file lock (mutex) that either side may acquire to block access from the other until it is released.

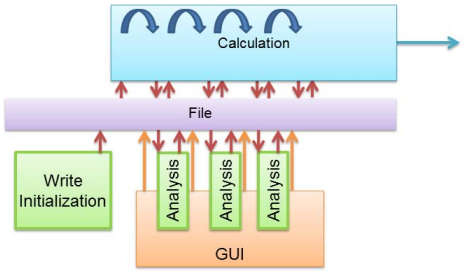
2.2. Operating Modes and Timing

It is assumed that the simulation will make regular/periodic writes to the file, and when using the steering API will issue reads to see if any new data or instructions are available. Since we do not support concurrent access to the file, a locking mechanism is required, along with a method to signal to the other side that new data has been written. After the simulation performs a write, it will close the file, releasing its lock and the file becomes available to the coupled process. At this point, two principal modes of operation are possible. The simulation may wait for its data to be further processed and some new commands or data to be returned, or it may

fire and forget making a quick check to see if anything has been left for it to act upon whilst it was calculating. The two modes of operation (`h5fd_dsm_set_mode`), referred to as *wait* mode and *free* mode are illustrated in figures 1(a) and 1(b).



(a) The simulation writes data periodically and waits for the analysis before continuing.



(b) The simulation loops iterations without waiting, the user interacts via GUI controls and new data is picked up whenever the simulation checks for it.

Figure 1: *Two principal modes of operation for timing of steering interactions.*

The illustration in figure 1(a) is self explanatory: after each iteration the simulation writes data and waits for the analysis task to signal that it is complete before the simulation reopens the file and collects new instructions and data. The *wait* operation is issued using a `h5fd_dsm_steering_wait` (see section 2.4) from the simulation, which then blocks until the next file handover by the analysis. *wait* mode can be considered as the most intuitive for a direct coupling of applications and will be used when a calculation explicitly depends upon a result of the analysis before it can continue, the actual amount of time the simulation waits will depend upon the workload/complexity of the analysis pipelines setup by the user.

In free mode, if the analysis is overlapped with the simulation and does not prevent it accessing the file then the simulation is normally delayed only by the time taken to check for new commands/data – which in the absence of any new instructions is of the order of milliseconds and for large simulations can be safely ignored. More detailed timing of bandwidths to/from the DSM compared to disk, and of the steering overhead can be found in [SB11] and [SBC10].

Note that although the diagram in figure 1(a) shows no user interaction taking place during the computation, the user interface is not blocked at this point and arbitrary operations may be performed by the user (including setup and initialization steps prior to the next iteration). Similarly, the calculation may perform multiple open/read/write/close cycles with different datasets prior to triggering an update and is not limited to a single access as hinted by the diagram.

Figure 1(b) shows a more complex example based on the *free* mode of operation. The calculation loops indefinitely issuing write commands, checking for new data using read commands and is permitted to open and close the file at any time (unless locked by the steering side). The simulation emits an update signal (via a call to `h5fd_dsm_server_udpate`) 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). Quite how the interaction between post-processing and simulation takes place is now entirely under the designer's control. A simulation that is operating in this free mode must be capable of receiving new commands/data and *know* that this data may not be directly related to the current calculation. At this point, the ability to send specific commands to the simulation that have special `meanings` becomes important. This is discussed further in section 4.

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 `h5fd_dsm_server_udpate` signal is received. This update is denoted by the periodically aligned green analysis boxes in figure 1. The user is free to modify the pipeline, change parameters and select outputs from it to be written back to the file. These GUI interactions will be semi-random and are denoted by the orange arrows in figure 1. The process is therefore entirely asynchronous and there are no restrictions on how the user may interact with the GUI and issue writes either with data or commands – it is the responsibility of the developer to ensure that the simulation can pick up data at a convenient point of the calculation. No events are triggered in the simulation, but the steering API provides routines to check if new commands have been received. The ParaView GUI however, does receive events triggered by the DSM service thread, which allows automatic pipeline updates. A final consideration is that whilst the *wait* mode may waste resources, and the *free* mode may be difficult to synchronize, the developer may switch to *wait* mode every N

iterations, to force some user interaction, then revert to *free* mode again for a period. Alternatively, the switch between modes may be user driven as a custom *command* (see section 2.4) and toggled by the user in the GUI.

2.3. System Configuration and Resource Allocation

It is clear from figure 1 that the amount of time/resources allocated to compute/steer tasks may have a significant impact on the overall performance of the system (particularly so in *wait* mode). For example, a simulation with very good scalability may be run on many cores, using a low memory per core and efficient communication, making good use of the HPC platform. The analysis required to control or steer the simulation may not scale well, or may require considerably more memory per node, but with less total cores – perhaps due to a very different pattern of access or communication. The DSM interface handles this by being quite flexible in how resources are allocated, consider figure 2, which shows general configuration types that may be used – the workflow can be distributed between different machines or set of nodes in a rather arbitrary manner. The first configuration, figure 2(a) corresponds to the most distributed arrangement where M nodes run the simulation code and N perform analysis. Tasks are coupled using the DSM in parallel – it is assumed that the network switch connecting machines has multiple channels so that traffic from M to N using the **inter-communicator** can take place in parallel and there is no bottleneck in communication. The final rendering stage can then happen on the same machine or on another machine (making use of the ParaView client/server framework [CGAF06]) or on the workstation where the ParaView client is running. Using separate machines makes it easy to ensure that optimized nodes (e.g. GPU accelerated) are used where needed.

If a hybrid machine is available, or if the simulation and analysis make use of similar node configurations, a single machine (c.f. figure (b)) may be used for both tasks – but note that *separate nodes* are used for the two tasks, so fine tuning of M and N is still permitted. Note also that whilst the default configuration of the DSM is to be hosted by the analysis task (server) on N nodes, the server may reside on either side of the **inter-communicator** link and thus be composed of M nodes. In this way (assuming $M > N$) either M small memory buffers, or N larger ones may be allocated, further enhancing the customization of the setup depending on the nodes/resources available.

Figure 2(c) shows the case where small data (or a very high end workstation) is under consideration, and all data can be analyzed on the workstation and commands sent back to the simulation.

2.4. Steering Interface

The main DSM interface is modeled after the existing HDF5 VFD drivers, with additional calls for our steering framework. The design of the original DSM driver was such that an existing HDF5 application could be visualized or post-processed *in-situ* by simply replacing the MPI-IO driver with the DSM one. Unfortunately, whilst passive visualization is straightforward, steering an application is not possible without some fundamental changes to the code. A brief overview of the steering API is presented here.

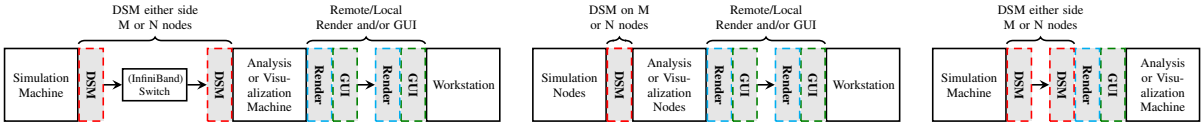
API

One of the first requirements when steering an application is the ability to change a simple scalar parameter. Since our API is built on top of HDF5, it is trivial to store such a parameter as an attribute within the file. Adding support for vectors requires only the use of a *dataset* in the file. Being memory based, the file write operations are cheap with no latency to disk, and being parallel in nature, any node of the client simulation may make a read of the parameter; the DSM VFD layer will retrieve it regardless of which server node it is actually placed on. Once the ability to write into an HDF5 *dataset* exists, it is easy to extend support to handle point arrays, scalar/vector arrays and all other `vtkDataArray` types that are used within ParaView to represent objects. We are thus able to write any structure to the file. One crucial factor is that both sides of the transaction must be able to refer to a parameter by a unique name, and find the correct value from the file. The developer is therefore required to assign unique names to all parameters and commands and use them in the simulation code. The steering environment is supplied these names in the form of an XML document which is described in section 3.1. The following `h5fd_dsm` commands are available:

```
(1)  h5fd_dsm_steering_init(comm)
(2)  h5fd_dsm_steering_update()
(3)  h5fd_dsm_steering_is_enabled(name)
(4)  h5fd_dsm_steering_scalar_get/set(name,mem_type_id,buf)
(5)  h5fd_dsm_steering_vector_get/set(name,mem_type_id,
       num_elem,buf)
(6)  h5fd_dsm_steering_is_set(name,set)
(7)  h5fd_dsm_steering_begin_query()
(8)  h5fd_dsm_steering_end_query()
(9)  h5fd_dsm_steering_get_handle(name,handle)
(10) h5fd_dsm_steering_free_handle(handle)
(11) h5fd_dsm_steering_wait()
```

In addition to the commands listed here, it is simple to *pause* and *resume* the controlled simulation, by locking and unlocking the file and therefore blocking the application at the next attempt to issue an HDF5 or `h5fd_dsm` command. These commands do not need to touch the contents of the *file* itself and are referred to as metadata commands.

By default all new parameters and arrays sent back for steering are stored at a given time step in an *Interaction* group which is a subgroup of the file (created automatically by the steering API layer). This group can be customized if necessary in case of conflict with the simulation data (in the event that it uses the same group name for data storage). One advantage of writing the interaction group directly into the HDF5 data is that a user may easily dump (using a DSM



(a) Simulation and analysis on separate machines, GUI client on a third.

(b) Simulation and analysis on the same machine, but using different nodes, GUI client separate.

(c) Simulation on one machine and all analysis/visualization on another.

Figure 2: *The DSM interface can be used in different configurations, between different machines or nodes of the same machine. Figure (b) may be the most commonly adopted as a local cluster may be treated as a simple extension of the workstation. Figure (a) is more likely when combining a highly optimized code on many cores with low memory, to a dedicated analysis cluster with fewer fat memory nodes. Figure (c) is more likely when the final data is smaller and can be handled on a high end workstation. Other combinations are possible if remote image delivery using a system such as VNC is considered.*

enabled `h5dump`) the parameters stored in order to check their presence or their correctness. In contrast with visualization only use of the DSM driver, for steering, the simulation needs to be able to read from the file at any time (including startup, for initialization data) and we therefore provide a steering library initialization call (1) which can be used to establish a connection between server and client before it would otherwise take place – the communicator used for IO is passed as a parameter if only a subset of nodes participate in IO. Once the environment is initialized, (2) allows the user to get and synchronize steering commands with the host GUI at any point of the simulation. (2) in effect is a special form of file close command which (in the ParaView plugin) also triggers pipeline updates in the GUI.

(4) and (5) allow the writing of scalar and vector parameters respectively, whilst (6) checks their presence in the file. As explained in section 2.2, parameters are dynamically written into the file so that one can get them at any time. The set/get functions are primarily for sending arrays from the GUI to the simulation, but may also be used by the simulation to send additional information to the GUI (such as time value updates at each step). Normally, all information would be present in the HDF5 output from the code anyway, but additional data may be passed in the *Interactions* group with convenient access using simple `h5fd_dsm_steering_vector_get` syntax – the sometimes tedious process of managing handles to file and memory spaces is taken care of by the API. As described in section 2.2, (11) can be used to coordinate the work-flow, making the simulation pause until certain steering instructions are received. User defined commands may be specified as booleans which are set after they are issued and then cleared, for example, a user defined command can be tested for and acted on as follows:

```
h5fd_dsm_steering_is_set("UserCommand", flag);
if (flag)
  PerformUserAction;
endif
```

(7), (8) are used when several consecutive operations are necessary. When accessed from the client side, file open and data requests result in **inter-communicator** traffic, which can be minimized by reducing HDF5 handle acquisition and release. Particularly when the file is open in read only mode, metadata is cached already by the underlying HDF5 library and traffic is correspondingly reduced.

(9) and (10) allow direct access to the HDF5 *dataset* handle to the requested object and this handle may be used with the conventional HDF5 API to perform IO. The advantage of this is that the full range of parallel IO operations may be used by making appropriate use of hyperslabs. This is particularly important if a very large array is modified by a user pipeline and returned to the simulation, where it must be read back in parallel on the compute nodes.

It is important to remember that the steering API commands listed above are intended as convenience functions for the exchange of *Interaction* data that would not normally take place. The standard HDF5 API should still be used for the bulk of data writes performed by the simulation for input to the steering application for analysis etc.

3. ICARUS ParaView plug-in

Up to this point, whilst the discussion has mentioned ParaView as the steering environment, there have been no ParaView specific modifications necessary. Any HDF5 based applications may be coupled together – with an implied assumption that one will be the master and the other the slave. In this section, we describe the enhancements we have made to the ParaView package to allow flexible creation of a customized steering environment.

A plug-in, called ICARUS (Initialize Compute Analyze Render Update Steer), has been developed to allow ParaView to interface through the DSM driver to the simulation. A significant portion of the work by a developer to use the plugin goes into the creation of XML templates which describe the outputs from the simulation, the parameters which may be controlled, and the inputs back to it. The XML description templates are divided in 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*.

3.1. Domain Description Template

Data read from HDF5 in our plugin makes use of the XDMF library for flexible import from a variety of sources – one of HDF5's great strengths is its ability to store data in many ways, but this in turn makes it difficult to know the layout of a particular simulation output without some help. We make use of XDMF as a convenience since it allows a simple description of data using XML (a customized HDF5 reader could equally well have been embedded in ParaView but would need to be configured individually for each simulation to be used). To read data (grid/mesh/image/...) one can either supply an XDMF description file as described in [CM07] or use an XML description template following the XDMF syntax, which our plugin uses, to generate a complete XDMF file on the fly. The XDMF template format we have created does not require the size of data-sets to be explicitly stated, only the *structure* of the data (topology/connectivity) needs to be specified with its path to the HDF5 data-set. As the file is received, the meta-data headers and self-describing nature of HDF5 data-sets allows the missing information (eg. number of elements in the arrays) to be filled-in (by in-memory routines using `h5dump`).

Visualization Properties

The template allows one or more *Grids* to be defined which are mapped to datasets in ParaView/VTK parlance. If the datasets written to the DSM are multi-block, as many grids as the number of blocks must be defined. Each *Grid* follows the following format example and contains at least a *Topology* field with the topology type, a *Geometry* field with the geometry type and the HDF5 path to access the data representing the geometry. Several attributes can then be added specifying for each the HDF5 path to access the data. Note that specific XDMF operations such as the *JOIN* can still be provided, for instance:

```
<Domain>
  ...
  <Grid Name="Particles">
    <Topology TopologyType="Polyvertex">
    </Topology>
    <Geometry GeometryType="X_Y_Z">
      <DataItem>/Step#0/X</DataItem>
      <DataItem>/Step#0/Y</DataItem>
      <DataItem>/Step#0/Z</DataItem>
    </Geometry>
    <Attribute AttributeType="Vector"
               Name="Velocity">
      <DataItem Function="JOIN($0, $1, $2)"
                ItemType="Function">
        <DataItem>/Step#0/VX</DataItem>
        <DataItem>/Step#0/VY</DataItem>
        <DataItem>/Step#0/VZ</DataItem>
      </DataItem>
    </Attribute>
    <Attribute>
      <DataItem>/Step#0/P</DataItem>
    </Attribute>
    <Attribute>
      <DataItem>/Step#0/Smooth</DataItem>
    </Attribute>
  </Grid>
  ...
</Domain>
```

The ICARUS plug-in generates from the template a complete (in memory) XDMF file with all the information about data precision and array sizes. When updates are received, the parallel XDMF reader extracts data directly from the DSM through the usual HDF5 operations. Note that only the ParaView client needs access to the template – the fully generated XML is sent to the server using the ParaView client/server communication.

3.2. Interaction Template

To define steering parameters, we follow the existing model of the ParaView server manager properties, which makes it possible to piggy back the automatic generation of controls on top of the existing mechanism used to generate filter/-source panels.

3.2.1. Steering Properties

1. Int/Double/String VectorProperty
2. CommandProperty
3. DataExportProperty

Int/Double/String VectorProperties allow scalar, vector and string parameters to be defined and generated in the GUI and are the same as the existing ParaView properties. Settings for default values, names, labels, etc, are available so that one can tidy up the automatically generated user interface. As with the ParaView server manager model, domains can be attached to these properties, this allows a user to restrict the parameters defined to either a boolean domain, which will be then seen as a check box, or to a range domain where possible input values are defined in a $[min;max]$ interval and appear as a slider.

Two new Properties have so far been added to support steering. One is a *CommandProperty*, represented in the GUI as a button, but without any state – when it is clicked, a flag of the defined name is set in the *Interactions* group and can be checked and cleared by the simulation (essentially a boolean without the ability to toggle on/off). An example may be seen in figure 4 and would be defined as follows:

```
<CommandProperty
  name="ReloadFreeBodyMesh"
  label="Reload free body mesh">
</CommandProperty>
```

A *DataExportProperty* defines an input back to the simulation, it allows a whole ParaView dataset or a single data array to be written into the file. One may interactively select a pipeline object, select the corresponding array (points,

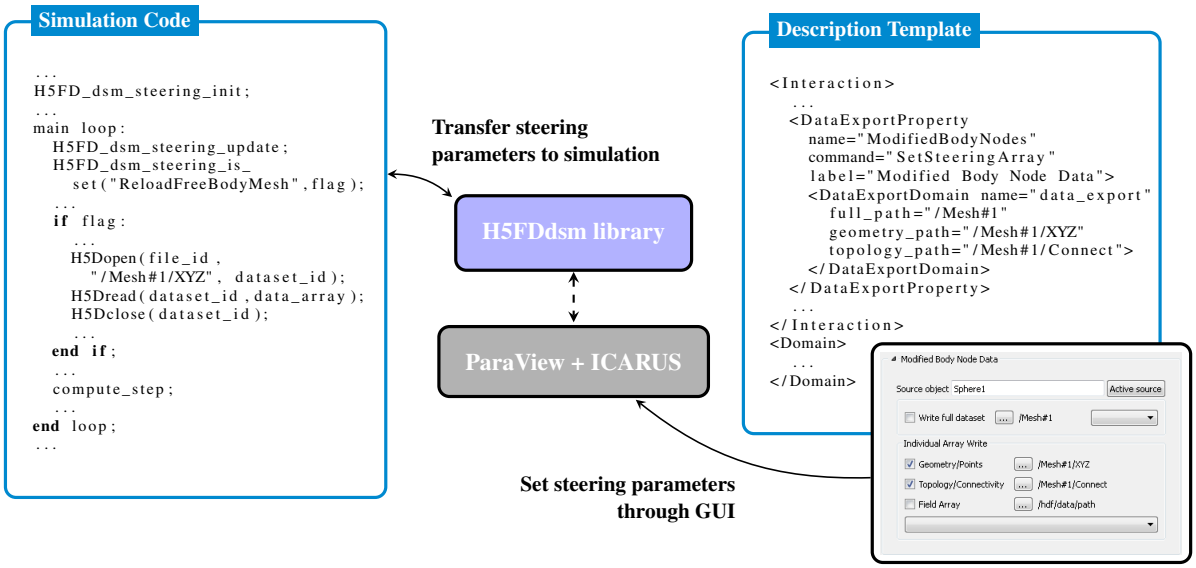


Figure 3: *Example of usage between a simulation code (SPH-flow), and ParaView – The user defines in a description template the interactions that the simulation will be able to access, GUI controls are automatically generated and modified parameters are passed to the H5FDdsm library. The simulation gets the parameters and the commands by reading them from the DSM using the same names as specified in the template.*

connectivity or node/cell field) and write it back to the DSM. The corresponding HDF path must be specified so that the location of the written array is consistent with the simulation's expectations. If the array is going to be a modified version of one sent initially to the GUI by the simulation, the user may reuse the path in which it was originally written to save space in the file. An example of the GUI generated is visible in figure 3.

If a grid exported by the simulation is to be modified directly – and then returned back to the simulation some action/control to be performed may be specified in the template and reference the grid in question. For example in section 4 with SPH-flow we wish to modify the geometry of the free body in the fluid, and we therefore bind a 3D interactive transform widget to it. This is done by adding hints to the properties (as below). Currently, any 3D widget may be created (box, plane, point etc), and for each grid with an attached widget, a mini-pipeline is created containing a series of filters, which extract the dataset from the multiblock input (if multiple grids exist) and bind the widget with associated transform to it. The GUI implementation and XML description are still experimental and we aim to add *Constraint* tags to the hints to specify that a grid may not be moved or deformed in some way, more than a specified amount per time step. A simulation may require certain constraints to prevent it causing program failure.

```
<Hints>
  <AssociatedGrid name="Body" />
  <WidgetControl name="Box" />
</Hints>
```

The mini-pipelines created to extract blocks are not exposed to the user, but do in fact make use of XML custom filters generated by ParaView itself. We plan to expose more of these internals to allow templates to be specified using hints, which contain complete analysis pipelines already enabled and embedded. The advantage of this is that no python scripting is required to set them up, and whilst full python scripting of all generated steering control and widgets is possible, this is not yet enabled. Note that the templates are loaded at run time and ParaView client/server wrappers for control properties (and mini pipelines) are generated on the fly – these are then registered with the server manager and objects instantiated – this means that all simulation controls can be created without any recompilation of either ParaView or the ICARUS plugin.

3.2.2. XML Steering Parser

One initial requirement when importing data from a simulation was the ability to turn off the export of data on a field by field or grid by grid basis. One does not wish to manually define a flag for each possible grid or field, so we make use of the generated XML file from the template that gives us access to all the information required to build a map of all the grids and arrays exported and display them in tree form in the GUI. This can be clearly seen in the right panel of figure 4. Each node of the tree can be enabled/disabled in the GUI with a corresponding flag allocated in the DSM metadata.

Two grids may have the same name, so we use the unique HDF path to name the flags. They can be read by the simulation (using `h5fd_dsm_is_enabled`) to tell it not to send a given grid or array. In this way we can reduce the network traffic to only the arrays we wish to work with for a particular analysis job without any recompilation or modification of code or template edits.

3.3. Time and Automated Steering

During development of the interface, it was evident that time management was a key concern. The simulation generates time steps that are displayed in ParaView, but we found that when working with SPH-flow – interactivity was sometimes a problem. By this we mean that when working with large data the simulation outputs data at a fairly slow rate (strong scaling is under development) and the user can spend some time waiting for data to arrive. Additionally we wished to modify grids in a smooth and continuous way which was not always possible using a mouse and 3D widget. For this reason we wished to use keyframe animation to move grids according to predefined paths, which could be adjusted on the fly. In order to animate cleanly, it was necessary to export at startup, the start and end times of the anticipated simulation, so that the keyframe editor could be initialized. To achieve this we send time range parameters at startup and at each step the current time. However time range updates and other non grid data sends caused trouble when the automatic update of pipelines took place. We therefore added an `update level` flag to the API using `h5fd_dsm_set_mode(H5FD_DSM_UPDATE_LEVEL_0+`N. This allows us to send information at startup using a `UPDATE_LEVEL_0`$+N$, where $N = 0$ is used for an information update, $N = 1$ for a pipeline update and $N \geq 2$ is available for custom messages. With these updates in place we were able to animate objects within the GUI and effectively use ParaView to generate geometry and send it to the simulation – which has no built in capability to produce meshes of it's own. Although mesh animation was desired principally, parameter animation linked to analysis is also possible and with this capability in place – combined with the ability to run in parallel – we believe a great many new applications will be found for this framework.

On each iteration the simulation (after checking for commands/data) will usually issue a file create command, which is used as a signal to the DSM to wipe and renew the contents. This prevents the memory file from growing ever larger as time progresses. If an analysis operation requires multiple steps in memory, then the simulation should add new data to the file in new datasets and leave N steps behind – either cleaning older datasets manually, or periodically performing a file create/wipe operation. A GUI control can be created to control this behaviour.

4. Application to SPH-flow

Several computational fluid dynamic models and particularly SPH models now make use of GPGPU for computing. This is for example the case of [GSSP10] where a very interactive simulation can be obtained and rendered using shaders or ray-tracing creating the effect of a real fluid. To obtain such a level of interactivity, precision and models must be less accurate and this is usually sufficient for creating visual effects. The solver we use here is designed for CPU computing and uses several different models providing a high degree of accuracy, which of course have the consequence that the more precision requested, the lower the interactivity. This solver, SPH-flow [OJG*09] is able to compute fluid and multi-physic simulations involving structures, fluid-structure, multi-phasic or thermic interactions on complex cases. The current version of SPH-Flow is mainly dedicated to the simulation of high dynamic phenomena, possibly involving complex 3D topologies that classical meshed based solvers cannot handle easily. A significant effort has been made to improve the SPH model towards more accuracy and robustness, together with high performance on thousands of processors.

Adding computational steering to SPH-flow did not require weeks of effort. Initially, simple calls to set the DSM driver were added, making it possible to monitor the data output of the code during runs. The original code computed and created a marching cube output which was written to a separate file, this unfortunately caused problems because by default, a file create in the DSM triggers a wipe of the memory (as opposed to a file open). Consequently, the second dataset written would remove the first. To solve this problem, calls to H5Fcreate were replaced by H5Fopen with checks to test if the DSM usage was enabled (as the code must still operate when not being steered). On start, calls to set the time range were inserted and after each iteration, a call to test if parameters are sent and calls to trigger updates were added. Modifying parameters such as the fluid inlet velocity became trivial as they required only simple `h5fd_dsm_steering_scalar/vector_get` calls. The largest part of the work has been to allow the code to reload a new geometry from the DSM when receiving a Reload command. This is because this capability did not exist before and so it represents an entirely new development and care must be taken that the new geometry does not appear in some way with uninitialized associated variables, causing the simulation to blow up. We have successfully steered the SPH-flow application using our entire local cluster, with 4 pvservers for visualization/analysis and 168 cores devoted to the simulation.

5. Related Work

The RealityGrid project [BCH*03], which is mainly used for grid computing allows the user to be connected dynamically to the simulation, monitoring values of parameters and

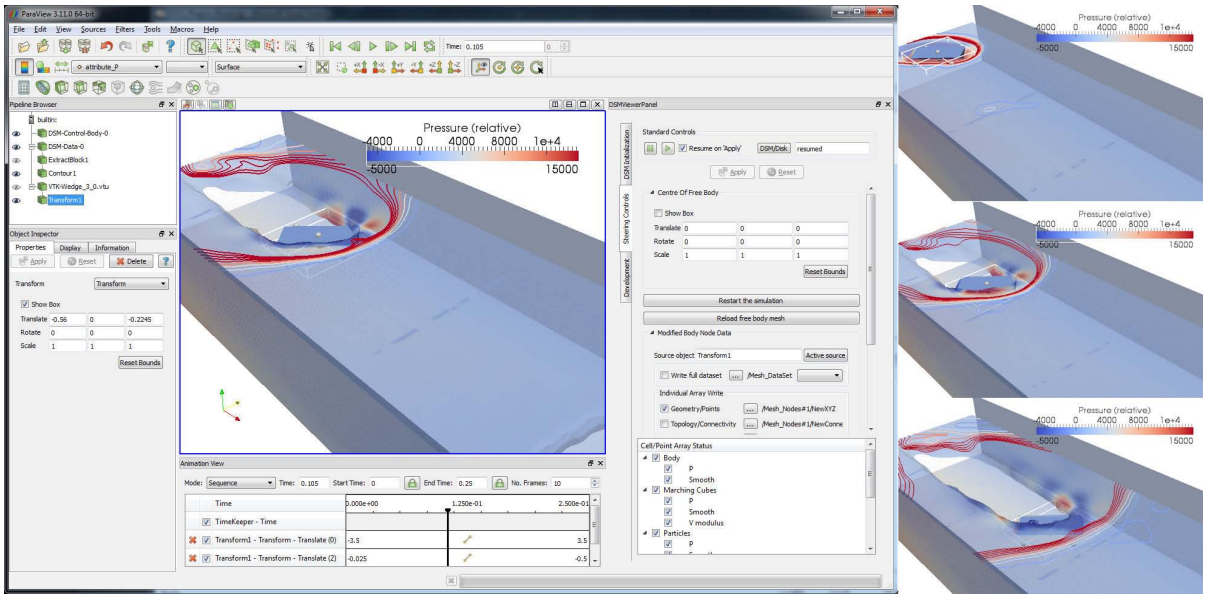


Figure 4: *The interface generated for SPH-flow using a template to describe 4 grids, one of which is to be controlled by a box widget. The right panel contains the generated GUI that is used to enter/modify parameters to control the simulation. The animation view (bottom) is setup to move the box widget through the domain and thereby drive the simulation. On the right are 3 snapshots of the simulation results as the mesh is pushed through the fluid – a scenario new to the simulation.*

editing them if necessary. Once a client is connected to the simulation component, it can send steering messages to the simulation, which transmits data to the visualization component. The computational steering API defined is quite exhaustive and provides many functions, however the necessary degree of intrusion inside the code is high. For a code already designed to take advantage of HDF5 IO, using our steering model coupled to the ICARUS plug-in is a significantly easier solution, reducing the likelihood of breaking a given code.

The EPSN [REC07] project defines a parallel high level steering model by manipulating and transferring objects such as parameters, grids, meshes and points. A user can ask for objects and these objects are automatically mapped (and redistributed) using the EPSN model to HDF5, VTK, or any other output format (for which a module in the library is provided – as is a ParaView plugin for visualization). One can then easily interface the simulation or visualization code to one of the mappers. As with the RealityGrid project, an interface allows registering steerable parameters and actions. The EPSN library also makes use of XML files to describe the data and also provides *task descriptions* that can be used to define synchronization points at which codes can wait for each other. We have taken many ideas from the EPSN development but found that for simple coupling the synchronization points are not necessary. As we move to more sophisticated scenarios these ideas may need to be revisited. EPSN includes a mesh redistribution layer which maps grids on N processes in one task to the M processes in the other, our system uses HDF5 as the parallel redistribution layer, leaving decision on how to partition data to the developer's original implementation. Additionally a simulation making use of EPSN must link to different high level library such as the VTK library as well as CORBA for thread management, whereas our simulation only requires the simulation to be linked against the HDF5 and MPI libraries.

VisIt [CBB*05] provides users with the libsim library, a lightweight library, which is portable enough to be executed on a large variety of HPC systems. The library provides an API so that one can interface his code to the VisIt environment. However it is necessary to re-work the code so that pointers to function loops can be passed to the interface. Whilst this mechanism is basically the same for every code, it does require a re-modeling of the simulation code and a knowledge of the interface. Whilst VisIt (libsim) and ParaView (coprocessing [MFMG10]) both provide in-situ visualization support, and both can be used for steering, the code is compiled and linked into the simulation. Furthermore, as described in [Chi07] and in [YWG*10] where a full in-situ visualization pipeline is applied on combustion simulations at large scale, making use of these in-situ visualization systems means that the analysis will run on the same cores as the simulation, placing additional memory demands on them and requiring them to wait for completion before resuming. In most cases, post-processing operations have to be well defined before running the simulation. The ability

for our library to use separate cores makes it more like the datastager [AWE*09] IO forwarding layer used by libraries such as ADIOS [LKS*08], though they do not yet support steering or coupling in the way our library does, such as allowing a complete or partial re-meshing of the boundaries as we have done with SPH-flow.

6. Conclusion and Future Work

We have presented a framework allowing an engineer or a scientist to enhance a parallel code using HDF5 extensions and XML templates so that it can communicate directly with a ParaView server and permit live visualization, analysis and steering. The system has a flexible allocation of resources on clusters or supercomputers and allows highly scalable simulations to interface to less scalable analysis pipelines without compromising the former. The underlying framework supports other types of coupling, which do not involve ParaView and could be used to couple two simulations, or mesh generators and simulations together using a shared virtual file.

On larger systems at CSCS such as the Cray XE6 using Gemini interconnect, it is not yet possible to join two applications dynamically using native MPI functions (without forthcoming vendor provided operating system/MPI layer fixes specific to the hardware/platform) and in the meantime, an additional mode of communication allowing us to run both the DSM server and the simulation on different nodes but within the same job is being developed. This solution involves launching both applications as part of the same job using a single communicator and will require modifications to simulation initialization code, but will permit the use of high speed RMA communication between tasks on large machines.

Acknowledgments

This work is supported by the *NextMuSE* project receiving funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement 225967.

The software described in this paper is available for download from `https://hpcforge.org/projects/h5fddsm` and `https://hpcforge.org/projects/icarus`.

References

[AWE*09] ABBASI H., WOLF M., EISENHAUER G., KLASKY S., SCHWAN K., ZHENG F.: DataStager: scalable data staging services for petascale applications. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing* (New York, NY, USA, 2009), ACM, pp. 39–48.

[BCH*03] BROOKE J., COVENEY P., HARTING J., JHA S., PICKLES S., PINNING R., PORTER A.: Computational Steering in RealityGrid. In *Proceedings of UK e-Science All Hands Meeting 2003* (2003), pp. 885–888.

[CBB*05] CHILDS H., BRUGGER E. S., BONNELL K. S., MEREDITH J. S., MILLER M., WHITLOCK B. J., MAX N.: A Contract-Based System for Large Data Visualization. In *Proceedings of IEEE Visualization 2005* (2005), pp. 190–198.

[CGAF06] CEDILNIK A., GEVECI B., AHRENS K. M. J., FAVRE J.: Remote Large Data Visualization in the Paraview Framework. In *Eurographics Symposium on Parallel Graphics and Visualization* (Braga, Portugal, 2006), Raffin B., Heirich A., Santos L. P., (Eds.), Eurographics Association, pp. 163–170.

[Chi07] CHILDS H.: Architectural Challenges and Solutions for Petascale Postprocessing. *Journal of Physics: Conference Series 78*, 1 (2007), 012012.

[CM07] CLARKE J. A., MARK E. R.: Enhancements to the eXtensible Data Model and Format (XDMF). In *HPCMP-UGC '07: Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 322–327.

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), SCA '10, Eurographics Association, pp. 55–64.

[GT07] GROPP W., THAKUR R.: Revealing the Performance of MPI RMA Implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Cappello F., Herault T., Dongarra J., (Eds.), vol. 4757 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 272–280.

[Hen05] HENDERSON A.: *ParaView Guide, A Parallel Visualization Application*. Kitware Inc., 2005.

[LKS*08] LOFSTEAD J. F., KLASKY S., SCHWAN K., PODHORSZKI N., JIN C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments* (New York, NY, USA, 2008), ACM, pp. 15–24.

[MFMG10] MORELAND K., FABIAN N., MARION P., GEVECI B.: *Visualization on Supercomputing Platform Level II ASC Milestone (3537-1B) Results from Sandia*. Tech. rep., Sandia National Laboratories, September 2010. SAND 2010-6118.

[OJG*09] OGER G., JACQUIN E., GUILCHER P.-M., BROSSET L., DEUFF J.-B., TOUZE D. L., ALESSANDRINI B.: Simulations of complex hydro-elastic problems using the parallel SPH model SPH-Flow. In *Proceedings of the 4th SPHERIC* (2009).

[REC07] RICHART N., ESNARD A., COULAUD O.: Toward a Computational Steering Environment for Legacy Coupled Simulations. In *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on* (July 2007), p. 43.

[SB11] SOUMAGNE J., BIDDISCOMBE J.: Computational Steering and Parallel Online Monitoring Using RMA through the HDF5 DSM Virtual File Driver. In *International Conference on Computational Science ICCS (to appear)* (2011).

[SBC10] SOUMAGNE J., BIDDISCOMBE J., CLARKE J.: An HDF5 MPI Virtual File Driver for Parallel In-situ Postprocessing. In *Recent Advances in the Message Passing Interface*, Keller R., Gabriel E., Resch M., Dongarra J., (Eds.), vol. 6305 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 62–71.

[YWG*10] YU H., WANG C., GROUT R., CHEN J., MA K.-L.: In Situ Visualization for Large-Scale Combustion Simulations. *Computer Graphics and Applications, IEEE 30*, 3 (2010), 45–57.