Parallel Computational Steering for HPC Applications Using HDF5 Files in Distributed Shared Memory

John Biddiscombe, Jerome Soumagne, *Student*, *IEEE*, Guillaume Oger, David Guibert, and Jean-Guillaume Piccinali

Abstract—Interfacing a GUI driven visualization/analysis package to an HPC application enables a supercomputer to be used as an interactive instrument. We achieve this by replacing the IO layer in the HDF5 library with a custom driver which transfers data in parallel between simulation and analysis. Our implementation using ParaView as the interface, allows a flexible combination of parallel simulation, concurrent parallel analysis, and GUI client, either on the same or separate machines. Each MPI job may use different core counts or hardware configurations, allowing fine tuning of the amount of resources dedicated to each part of the workload. By making use of a distributed shared memory file, one may read data from the simulation, modify it using ParaView pipelines, write it back, to be reused 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 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 (and inputs back to it) to automatically generate GUI controls for manipulation of the simulation.

Index Terms—Parallel I/O, distributed/network graphics, software libraries.

1 Introduction

Scientists and engineers are continually improving the capabilities of their simulation codes to take advantage of the ever growing processing power of both modern HPC resources and desktop machines. As codes evolve from the desktop onto larger machines it is natural for the developer to want to continue to experiment with parameters 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 rapid feedback on the effects they have.

The motivation for the work presented in this paper has been 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 remeshing operations while the simulation continues, is required. A principal requirement of our steering environment is that it be flexible enough to interface to a variety of codes using different programming languages and different data models. Additionally, it should be possible to generate GUI controls to drive simulations, without extensive work by the domain scientists who wish to concentrate their efforts on the simulation code. For this reason, a solution involving an existing application—in this case ParaView [1]—capable of parallel analysis and providing full visualization features and using a client/server architecture for remote connections, was sought.

The implementation we present here is built around the concept of a file in Distributed Shared Memory (DSM) which can be read/written by the simulation, and asynchronously (but not concurrently) by the analysis code. Data written into the DSM are described using simple XML templates which tell the analysis where to find the objects it requires and how it is arranged in memory (dimensions, hierarchies, and shapes of objects). The XML is also used to tell the GUI component how to present controls to the user and what objects may be modified.

In the following sections, we describe the design and implementation of the framework that has been presented previously in [2], but include results enabled by a static connection mode of MPI tasks, which makes it possible to use our system on large HPC machines. We also include a description of improved synchronization and notification

- J. Biddiscombe, J. Soumagne, and J.-G. Piccinali are with the CSCS Swiss National Supercomputing Centre, Galleria 2, Via Cantonale, Manno 6928, Switzerland. E-mail: {biddisco, soumagne, piccinal}@cscs.ch.
- G. Oger and D. Guibert are with the Ecole Centrale de Nantes, Nantes 44300, France. E-mail: {guillaume.oger, david.guibert}@ec-nantes.fr.

Manuscript received 1 Sept. 2011; revised 23 Dec. 2011; accepted 13 Jan. 2012; published online 13 Feb. 2012.
Recommended for acceptance by R. Pajarola and K. Zhou.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2011-09-0213.
Digital Object Identifier no. 10.1109/TVCG.2012.63.

mechanisms, simplifying the handshaking between client and server and providing push-driven data transfers.

The paper is organized as follows: we first discuss related work in Section 2, then in Section 3 we discuss the I/O coupling, the architecture and communicators used to achieve high bandwidth data transfers as well as the operating modes, synchronization and notification required for handshaking. Section 4 outlines the development of a plug-in for ParaView and how the simulation description is created and interpreted to produce a controlling environment. Section 5 deals with a specific application example where an SPH code has been coupled and controlled by ParaView.

2 Related Work

Steering of HPC applications has been extensively treated by the RealityGrid project [3], which (as the name suggests) is principally intended for grid computing. The framework allows the user to be connected dynamically to the simulation, monitoring values of parameters and editing them if necessary. Once a client is connected to the simulation component, it can send steering messages to the simulation, which in turn 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 as are the number of libraries used. One interesting aspect of the RealityGrid project is an emphasis on steering of very slow applications, which may in fact be extremely *expensive* from a computational point of view. This makes it important that the status be monitored from time to time so that time consuming calculations may be stopped if the simulation is not evolving in the desired way. Within RealityGrid, where applications have been enhanced to support it, one may stop the simulation, go back to a previous restart, change parameters and continue, potentially exploring a number of regions in parameter space, always retaining the ability to stop, go back and retry a new or earlier path.

Our solution differs somewhat in design from RealityGrid as we require only the use of a single HDF5 library and file driver extension, so applications can be modified to make use of it very quickly. We focus on simulations where the results are arriving at a rate where the user might wish to interactively change parameters and we directly couple the analysis/visualization bidirectionally in such a way that the analysis may compute derived data sets and return them to the simulation, thus driving quite complex changes in the evolution of the simulation.

The EPSN [4] project defines a parallel high level steering model allowing manipulation and transfer of objects such as parameters, grids, meshes, and points. A code can request objects using the EPSN data model and these objects are automatically mapped (and $M \times N$ redistributed) to HDF5, VTK, or any other output format for which a module in the library exists (a ParaView plug-in for visualization using the VTK mapping exists). One can interface the simulation or visualization code to one of the mappings by decorating data structures using wrappers around them which make them *visible* to the EPSN library, effectively reinterpreting the original data model in terms of the EPSN model. As with the RealityGrid project, an interface allows registering steerable parameters and actions. The EPSN library 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.

EPSN includes a mesh redistribution layer that 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 (on either side of the coupled applications) 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 object management, whereas our simulation only requires the simulation to be linked against the HDF5 (and MPI) libraries.

VisIt [5] provides users with the libsim [6] in-situ visualization library, a lightweight library, which is portable enough to be executed on a large variety of HPC systems and with an API that one can simply interface to the VisIt environment. ParaView provides a (coprocessing [7]) library that allows a simulation to make function calls to pass its current solution state to a ParaView coprocessor. The coprocessor then reads instructions from a Python script and builds a filter pipeline for analysis of the data.

Both libraries require some reworking of the code so that pointers to data can be passed to the interface, and while the interface conversion is basically the same for each simulation code (and similar to that of most I/O libraries), it does require a detailed knowledge of the simulation and visualization tool interface/workings. These in-situ libraries can be used for limited steering of the simulation, but fall short of the capabilities provided by our bidirectional implementation, which allows the sending back of modified data sets. Additionally, one major drawback of tightly coupled in-situ libraries such as these is 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, postprocessing operations have to be well defined before running the simulation. The ability for our library to use separate cores (loosely coupled) makes it more like the DataSpaces method [8] defined in ADIOS [9], though it does not yet support steering in the way our library can, but does permit the separation of simulation and analysis, by creating a virtual shared memory space, i.e., a staging area that can be asynchronously accessed using one-sided communication protocols. Multiple time steps can be stored in this staging area and are automatically deleted depending on space demands or user requests. Advanced mapping and advanced redistribution mechanisms using PGAS models are also being developed [10]. Our system is somewhat simpler to configure since it uses only a single HDF5 file library and driver and also offers bidirectional transport intended for steering as well as postprocessing.

A similar approach is provided with GLEAN [11]. The framework uses a client/server architecture to reroute data from a simulation to staging nodes. In this approach, the simulation is the client and the staging part receiving data from the client is the server, i.e., the client runs on compute nodes or on dedicated I/O nodes and the server runs on staging or visualization nodes that are connected to the compute nodes via a local network. These frameworks (such as DataSpaces and GLEAN) are principally intended to accelerate I/O thereby improving scalability of supercomputing applications, they are currently being improved with postprocessing capabilities to reduce data on the fly and thus reduce the overall I/O burden.

3 DSM Interface

In [12], an approach was presented to in-situ postprocessing using a DSM file as the interface between arbitrary simulation and postprocessing applications that use the HDF5 API for the exchange of data. HDF5 supports virtual file driver (VFD) extensions that allow the customization of I/O 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 reroutes 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/postprocessing operations as desired. The original design presented in [12] catered only for write operations by the simulation, followed by reads from the host application, but recent work has extended it (see [13]) to support bidirectional read/write accesses using a file lock to block access from one side while the other is using it.

3.1 Architecture

The DSM uses a client/server model where (generally) the simulation writing the data is the client and the set of (postprocessing) nodes receiving the data is the server.

3.1.1 Communicators

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 intracommunicator and intercommunicator are used:

1. An **intracommunicator** represents the communicator used for internal communications by a given application or job, this communicator always uses the MPI interface;
2. An **intercommunicator** 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 **intercommunicator**. When used with ParaView, this means that the *pvserver* tasks each have an extra thread which is waiting to serve put/get requests from the simulation—the simulation tasks do not suffer any such penalty. With multicore processors becoming the norm and memory per core being

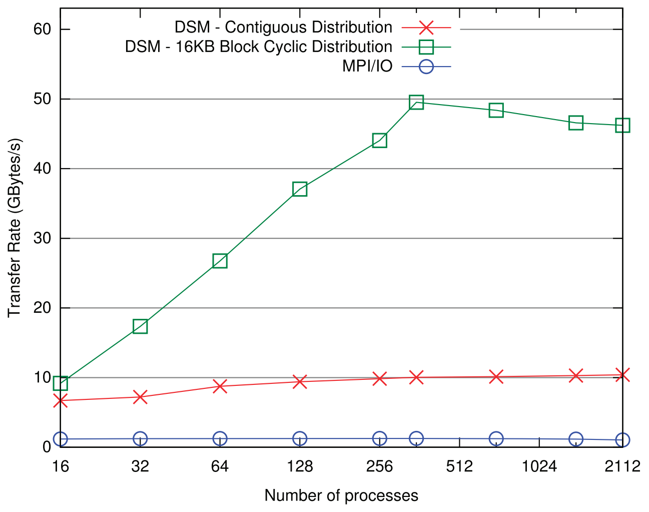


Fig. 1. Write transfer rate on a Cray XE6 of a 44GB file (10 data sets) to a DSM distributed among 88 nodes (176 processes) using MPI RMA.

limited, we have found that an extra thread per *pvserver* does not impact our usage as we require more memory to host the DSM and receive data. Note that in fact when using a GUI-based application such as ParaView with the DSM server, we also use a notification thread (on process rank 0) to signal the GUI that data are ready, but this thread is left suspended except when triggering events in the GUI as explained in Section 3.1.3.

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 and therefore directly access remote memory regions. In addition, when RMA is used, the DSM synchronization mechanism (which is 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 are written directly into the server side memory. For these reasons, the RMA method is the preferred protocol, even though MPI 2 one-sided API is still limited for implementing distributed shared address space models, and our model should take advantage of the upcoming MPI 3 API. The latest specialized architectures developed for HPC increasingly make use of MPI implementations that support RMA operations at the hardware level and give good performance [14] in terms of raw throughput of data.

Additionally as presented in [15], we recently enhanced the DSM redistribution methods so that data pieces are automatically remapped to follow a block cyclic redistribution pattern which stripes data across server nodes and increases throughput by including more network links for each large data write. Data transfer results for a bandwidth test application are shown in Fig. 1 using a Cray XE6 machine, one can easily see the scaling limitation of MPI disk I/O compared to DSM I/O. For this test, the number of processes per node is kept constant until all the nodes available on the machine are utilized (which happens at 352 processes), this represents the point where all network links on the machine are being used. From 353 to 2,552 processes the cores per node

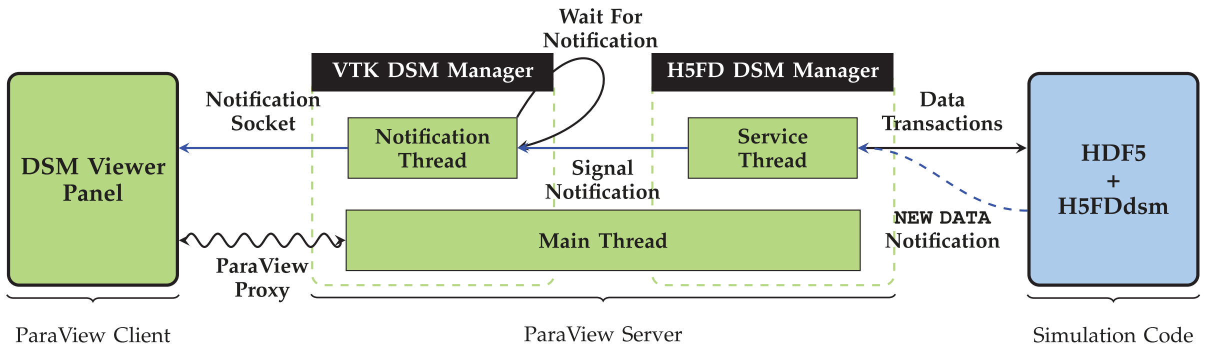


Fig. 2. Thread and push-driven notification mechanism used to inform the GUI and the DSM of new events. When a new data or new information notification is received, the task relative to this event is performed and/or the associated ParaView pipeline is updated (demand-driven).

writing are gradually increased until reaching the maximum number of cores available per node (24). We see the bandwidth at this point flatten as the network is being saturated. Unlike traditional I/O nodes, the number of DSM nodes can be easily increased depending on the needs of the current job—as more network links are used, as higher the bandwidth is, so there is no fundamental limit to the bandwidth we might achieve other than the machine specifications. It is worth noting that in the extreme case (on a torus network topology) of using half the nodes on one side of the machine for simulation and half on the other as DSM servers, then the achievable read or write bandwidth calculation would be effectively half the bisection bandwidth of the machine, however one would prefer to distribute the DSM nodes more randomly to avoid the bottlenecks of shared links if possible—we have not yet explored all the possible node placement configurations. To create the timing plot of Fig. 1, it was necessary to add a new connection mode to the DSM driver, a *static* MPI connection using an MPMD job launch where the simulation and analysis are both run as part of the same MPI job. This is because systems such as Cray/IBM leadership class machines do not currently support the dynamic process management family of functions. When using this *static* connection mode, it is necessary to replace the `MPI_COMM_WORLD` communicators in both simulation and analysis with a split communicator, and this was necessary for ParaView and SPH-Flow in order to generate the plot of Fig. 8 in Section 5.

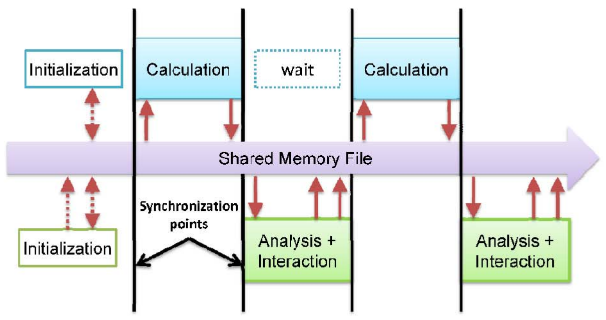
3.1.2 File Consistency

Any operation, which modifies metadata in the file (such as layout changes of data sets/attributes in the file), flags the driver that a synchronization step must take place prior to subsequent file data 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 I/O within HDF5 requires collective operations between all nodes using the same file when these metadata changes are made: providing the client or the server use the file independently, the HDF5 layer will handle local synchronization, but if both sides were to attempt to read/write concurrently, an additional exchange of metadata would be required between tasks which 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. After the simulation finishes writing, it will close the file, releasing its lock and the file becomes available to the coupled process.

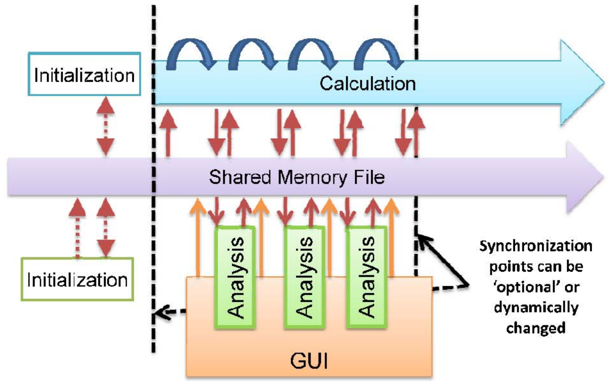
3.1.3 Notification Mechanism

It is assumed that the simulation will make 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 notify the other side that new data have been written. The notification mechanism is illustrated in Fig. 2. In a pull-driven system, the analysis code must query whether new data are present and if so, update its analysis pipelines—this would require a polling operation driven from the ParaView GUI causing the *pvserver* instances to check for new data. This may not be possible if some user related task is currently running and would block other GUI/server requests if they were taking place.

Instead we use a push or event driven approach to notify the GUI when new data are produced. It works in several stages as follows: Each DSM server process has a constantly listening *Service Thread*, which receives and treats internal data transactions, lock commands and eventually data itself if the communication model chosen is not one-sided RMA. When the file is closed by the simulation (or if a specific notification request using a `h5fd_dsm_notify` call has been added to the code), a notification is sent to the DSM server and is picked up immediately by the *Service Thread*. A *Notification Thread* is now woken on rank 0 to send a notification event to the GUI via a socket connection (which does not interfere with the normal ParaView client/server socket-based communication). When the GUI is notified, the task corresponding to the received notification code is then performed in the DSM user interface. If this is a data (as opposed to information) update, then the GUI schedules the execution of the user instantiated pipelines on the usual client application thread, and from there back into the *pvserver* main thread where the pipeline execution actually takes place. This two step process from (DSM) server to GUI and back to (ParaView) server again ensures that the we do not trigger the execution of pipelines during another user driven event on the same thread inside the *pvserver* tasks. Note that the user may add as many data write/close and notification calls to their code as required and disable automatic notification whenever the file is closed by using



(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.

Fig. 3. Two principal modes of operation for timing of steering interactions.

h5fd_dsm_set_options—this gives flexibility to how the user interaction is arranged.

It is important to note also that the notification mechanism is one sided only—when ParaView writes data to the DSM, no signals are triggered in the simulation as there is no *Service Thread* running on the DSM client side, the simulation simply requests the lock when it wishes to load/check for new data—if the analysis is still using the DSM then the simulation will block—this is to be avoided by ensuring that the analysis completes before the simulation completes its iteration loop.

3.1.4 Operating Modes

With the notification structure in place, 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 while it was calculating. The two modes of operation, referred to as *wait* mode and *free* mode are illustrated in Figs. 3a and 3b.

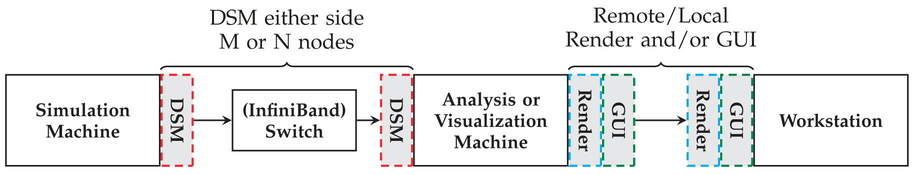
The illustration in Fig. 3a is self-explanatory: after each iteration the simulation writes data and waits for the analysis task to complete before the simulation reopens the file and collects new instructions and data. The *wait* operation is issued using a h5fd_dsm_steering_wait (see Section 3.3) from the simulation, which then blocks until the next file handover by the analysis. The *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 set up 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 as it would generally take place at a point where I/O operations would occur anyway. As noted in Section 3.1.3, it is important to ensure that the analysis does not hold the file lock for longer than the wall time of one of the simulation iterations. Usually, data will be read, the file unlocked immediately after and analysis will take place asynchronously, but if data must be written back before the iteration completes, it is best achieved by increasing the number of nodes dedicated to the analysis part of the operation (assuming good scalability).

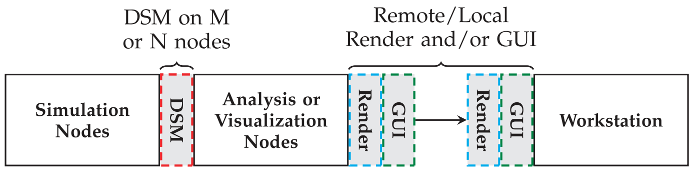
Note that although the diagram in Fig. 3a shows no user interaction taking place during the computation, the user interface is not blocked at this point and arbitrary operations may be performed by the user (including setup and initialization steps prior to the next iteration). Similarly, the calculation may perform multiple open/read/write/close cycles with different data sets prior to triggering an update and is not limited to a single access as hinted by the diagram.

Fig. 3b 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 automatically emits notification signals on file close when notification is enabled (and manually via a call to h5fd_dsm_notify) whenever it has completed a step 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 postprocessing time step T while the simulation has begun computing $T+1$ (assuming that we are talking about a simulation that iterates over time). Quite how the interaction between postprocessing 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 5.

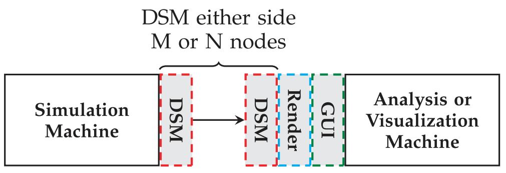
While the simulation is calculating, the steering side is free to perform analysis, modify parameters, and write new data to the file. Usually, there will be a fixed pipeline setup in advance to slice, contour, etc., and render the data as soon as a DSM notification signal is received. This update is denoted by the periodically aligned green analysis boxes in



(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.

Fig. 4. 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 are smaller and can be handled on a high end workstation.

Fig. 3. 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 semirandom and are denoted by the orange arrows in Fig. 3. 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.

A final consideration is that while 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 3.3) and toggled by the user in the GUI. This flexibility allows the user to let the simulation run freely for a while, enable *wait* mode when wanting to change something and then have the simulation pick up new data and go back to *free* mode until the next time a change seems necessary.

3.2 System Configuration and Resource Allocation

It is clear from Fig. 3 that the amount of time and resources allocated to compute or 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 Fig. 4, which shows general configuration types that may be used—the work flow can be distributed between different machines or set of nodes in a rather arbitrary manner. The first configuration, Fig. 4a 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 **intercommunicator** 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 [16]) 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 nodes) 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 (cf. Fig. 4b) 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 while the default configuration of the DSM is to be hosted by the analysis task (server) on N nodes, the server may actually reside on either side of the **intercommunicator** 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 resources available.

Fig. 4c shows the case where small data (or a very high end workstation) are under consideration, and all data can be analyzed on the workstation and commands sent back to the simulation.

3.3 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 postprocessed *in-situ* by simply replacing the MPI-IO driver with the DSM one. Unfortunately, while passive visualization is straightforward, steering an application is not possible without some fundamental changes to the code to respond appropriately to changes being sent in. A brief overview of the steering API is presented here.

3.3.1 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 *data set* 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 *data set* 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 shared/steerable parameter or data set 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 4.1.

The following 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()
```

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). A convenient side effect of writing interactions directly into the HDF5 data is that a user may at runtime easily display in text form (using a DSM enabled `h5dump`), the parameters/structures stored in order to check their presence or their correctness. In contrast with a visualization only use of the DSM driver, when steering, the simulation needs to be able to *read* from the file at any time (including at startup, for initialization data) and we therefore provide a steering library initialization call command 1 which can be used to establish a connection between server and client before it would otherwise take place (at file creation)—the communicator used for I/O is passed as a parameter if only a subset of nodes participate in I/O. Once the environment is initialized, command 2 allows the user to get and synchronize steering commands with the host GUI at any point of the simulation. Command 2 in effect is a special form of file close command which (in the ParaView plug-in) also triggers pipeline/other updates in the GUI.

Commands 4 and 5 allow the writing of scalar and vector parameters, respectively, while command 6 checks their presence in the file. The set/get functions are primarily used for sending arrays from the GUI to the simulation, but they may also be used by the simulation to send additional information to the GUI (such as time value updates at each step). Normally, one would expect all such information to be present in the HDF5 output from the code anyway, but additional data may be passed in the *Interactions* group with convenient access using the unique names and 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 3.1.4, command 11 can be used to coordinate the work flow, making the simulation pause until certain steering instructions are received. Additionally, it is possible to forcefully *pause* and *resume* the controlled simulation, by locking and unlocking the file from the GUI side, thereby blocking the application at the next attempt to issue an HDF5 open or `h5fd_dsm` command. The use of command 11 is preferred as it offers the chance to add wait/resume matching pairs of calls to the codes at arbitrary positions where the simulation should automatically pause to pick up new instructions.

User defined commands may be specified as booleans which are set after they are issued and then cleared after access, for example, a user defined command can be tested for and acted on as follows:

```
h5fd_dsm_steering_is_set("UserCommand",flag);
if (flag)
  /* Perform User Action */
endif
```

Commands 7 and 8 are used when several consecutive operations are necessary. When accessed from the client side, file open and data requests result in **intercommunicator** 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.

Commands 9 and 10 allow direct access to the HDF5 *data set* handle to the requested object and this handle may be used with the conventional HDF5 API to perform I/O. The advantage of this is that the full range of parallel I/O operations may be used by making appropriate use of hyperslabs. This is particularly important if a very large array is modified (in parallel) by a user pipeline and returned to the simulation, where it must in turn 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. In fact the standard HDF5 API can be used to access most of the interactions mentioned, but the steering API makes the process much simpler.

4 Initialize Compute Analyze Render Update Steer (ICARUS) ParaView Plug-in

While the discussion has mentioned ParaView as the steering environment, 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, 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 plug-in 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*.

4.1 Domain Description Template

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. The eXtensible Data Model and Format (XDMF) has been designed toward that goal [17], by providing users with a comprehensive API and data description format based on XML, the eXtensible Markup Language. It has been developed and maintained by the US Army Research Laboratory (ARL), and is used by several HPC visualization tools such as ParaView, VisIt, etc. XDMF distinguishes heavy data, significant amount of data stored using HDF5, from light data, smaller amount of data where values (typically less than about a thousand) can be passed using XML. Data are organized in *Grids*, which can be seen as a container for information related to 2D and 3D points, structured or unstructured connectivity, and assigned values. For each *Grid*, XDMF defines topology elements (Polyvertex, Triangle, Tetrahedron, ...), which describe the general organization of the data, and geometry elements, which describe how the XYZ coordinates are stored. Attributes (associated values to the mesh) can be described as well as *Grid* collections, which enable addition of time information.

Data read from HDF5 in our plug-in makes use of the XDMF library for flexible import from a variety of sources and 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 be 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 [17] or use an XML description template following the XDMF syntax, which our plug-in 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 metadata headers and self-describing nature of HDF5 data sets allows the missing information (e.g., number of elements in the arrays, precision, dimensions) to be filled-in (by in-memory routines using `h5dump`).

4.1.1 *Visualization Properties*

The template allows one or more *Grids* to be defined, which are mapped to data sets in ParaView/VTK parlance. If the data sets written to the DSM are multiblock, 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 to combine fields into vector arrays, 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>
```

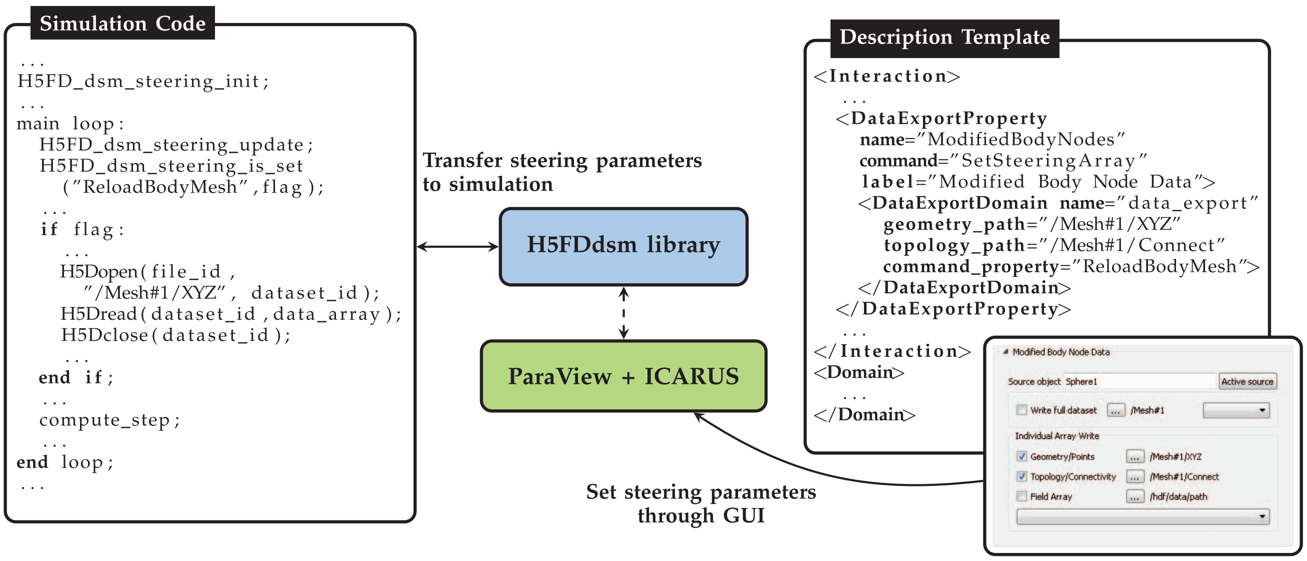
The ICARUS plug-in generates from the template a complete (in memory) XDMF file with all the information about data precision and array sizes. When updates are received, the parallel XDMF reader extracts data directly from the DSM through the usual HDF5 operations. Note that only the ParaView client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

4.2 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.

4.2.1 *Steering Properties*

Int/Double/String VectorProperties allow scalar, vector, and string parameters to be defined and generated in the GUI and are exactly the same as the existing ParaView properties. Settings for default values, names, labels, etc., are available so that one may tidy up the automatically generated user interface. As with the ParaView server manager model, domains can be attached to these properties, this allows a


Simulation Code
Transfer steering parameters to simulation
H5FDdsm library
ParaView + ICARUS
Set steering parameters through GUI
Description Template

```
...
H5FD_dsm_steering_init;
...
main loop:
  H5FD_dsm_steering_update;
  H5FD_dsm_steering_is_set
    ("ReloadBodyMesh",flag);
  ...
  if flag:
    ...
    H5Dopen(file_id ,
      "/Mesh#1/XYZ", dataset_id);
    H5Dread(dataset_id ,data_array);
    H5Dclose(dataset_id);
    ...
  end if;
  ...
  compute_step;
  ...
end loop;
...
```

```
<Interaction>
  ...
  <DataExportProperty
    name="ModifiedBodyNodes"
    command="SetSteeringArray"
    label="Modified Body Node Data">
    <DataExportDomain name="data_export"
      geometry_path="/Mesh#1/XYZ"
      topology_path="/Mesh#1/Connect"
      command_property="ReloadBodyMesh">
    </DataExportDomain>
  </DataExportProperty>
  ...
</Interaction>
<Domain>
  ...
</Domain>
```

Modified Body Node Data
Source object Sphere1 | Active source
☐ Write full dataset ... /Mesh#1
Individual Array Write
☑ Geometry/Points ... /Mesh#1/XYZ
☑ Topology/Connectivity ... /Mesh#1/Connect
☐ Field Array ... /hdf/data/path

Fig. 5. 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.

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 using a $[min; max]$ interval and appear as a slider.

<IntVectorProperty> .... </IntVectorProperty>
<DoubleVectorProperty> ..... </DoubleVectorProperty>
<StringVectorProperty> ..... </StringVectorProperty>
<CommandProperty> .... </CommandProperty>
<DataExportProperty> .... </DataExportProperty>

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 and is used to tell the simulation to perform some user defined action. It 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 data set or a single data array to be exported into the file. One may interactively select a pipeline object, select the corresponding array (points, 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 Fig. 5. A data export property is generally bound to a command property so that data are written and a command to perform some action with it are sent together.

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 5 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 instantiated containing a series of filters, which extract the data set from the multiblock input (if multiple grids exist) and bind the widget with an associated transform to it. The GUI implementation and XML description are continually improving 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 such as preventing an object being moved by more than a defined amount (such as a CFL condition) or smoothly (continuously differentiable). Constraints might even be derived from computed values combined with other parameters.

```
<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. Note that the templates are loaded at runtime 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 plug-in.

4.2.2 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. Each node of the tree can be enabled or 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 send the arrays we wish to work with for a particular analysis job without any recompilation or modification of code or template edits.

4.3 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 continuous development) and the user can spend some seconds waiting for data to arrive. 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 and deform 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 correctly. To achieve this we send time range parameters at startup and at each step the current time to drive the ParaView animation controls.

However, time range updates (and other *informational* data receipts) triggered the automatic update of pipelines setup for analysis which were invalid, so specialized user notification flags passed using `h5fd_dsm_notify()` were required in the API. Initially, it was necessary to distinguish only new *information* from new *data*, so the following flags have been added: `H5FD_DSM_NEW_INFORMATION` and `H5FD_DSM_NEW_DATA`. This allows us to send information at startup with `H5FD_DSM_NEW_INFORMATION`, querying an information update, and then large data with `H5FD_DSM_NEW_DATA` for a pipeline update—user defined notification keys may be sent from the simulation, and user defined event handlers may be added to the GUI to perform custom actions on their receipt.

With these updates in place we are 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. One such example already in development is the calculation of erosion on a surface to produce a deformed surface which can be written back to the simulation and in turn modify the fluid flow. This can be performed as a postprocessing step, which interacts with the simulation as it is not critical that deformations are sent on every iteration, but only when a significant change has built up.

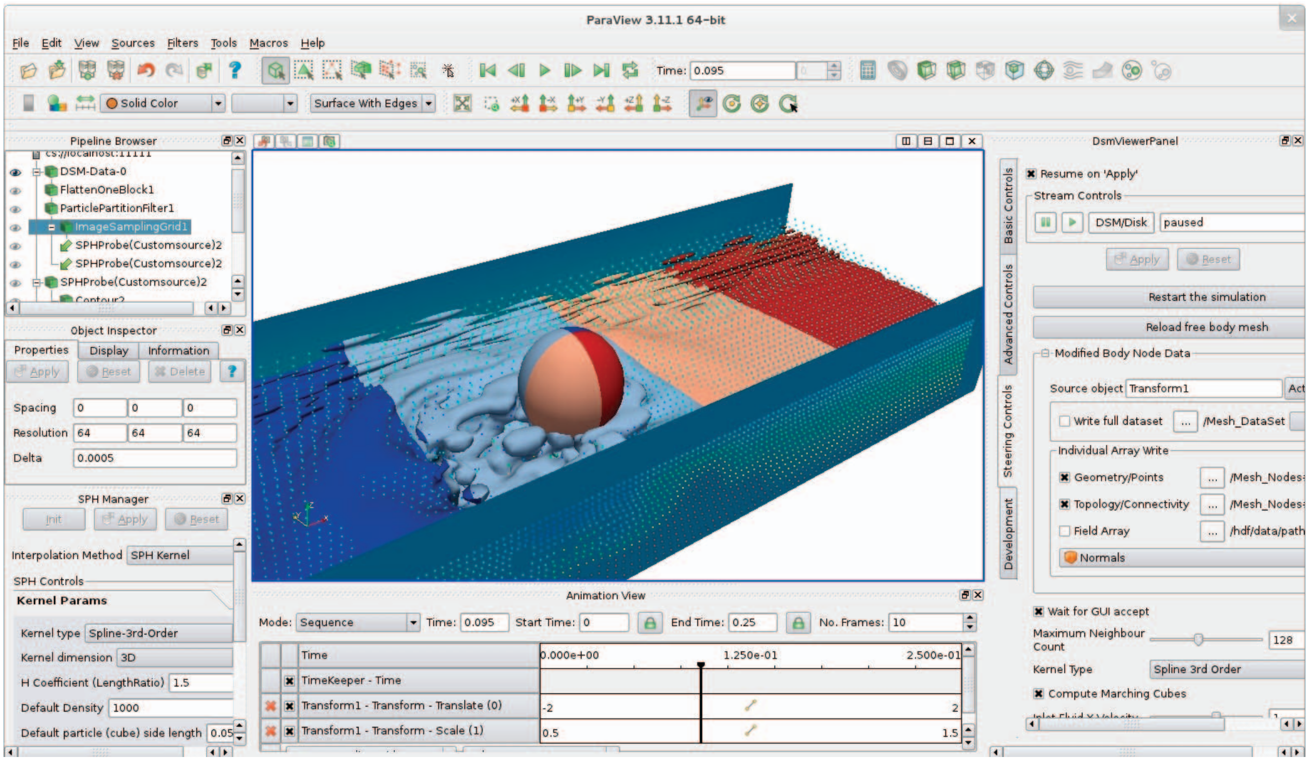
Most simulations write a new file on each iteration—this file create operation is taken as a command to cleanly wipe the DSM and thus prevent memory usage from growing ever larger as each time step accumulates. If the analysis requires multiple time steps to be preserved, the file create can be swapped for a file open and then successive time steps build up inside the DSM (providing sufficient space was allocated initially). This behavior can of course be placed under user control by adding XML commands to decorate the GUI so that the frequency of open/create calls may be changed or forced.

5 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 [18] where a highly interactive simulation is 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 but 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 [19] is able to compute fluid and multiphysic simulations involving structures, fluid-structure, multiphasic 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 toward more accuracy and robustness, together with high performance on thousands of processors.

Adding simple visualization and computational steering (parameters only) to SPH-flow did not require significant effort. Initially, simple calls to set the DSM driver were added, making it possible to monitor the data output of the code during runs. Some small changes were made to the code to reorder certain write operations and combine multiple outputs into a single file rather than multiple files (each new file would trigger a wipe of the DSM rather than adding to the existing data). Additional information writes for time and notifications and wait/resume requests were then added which allowed the simulation to be viewed and stopped/started under user control. Simple scalar parameter changes could be made with double/int properties and propagated back into the simulation with simple `h5fd_dsm_steering_scalar/vector_get` calls for parameters such as the inlet boundary velocity of the fluid.

To make more significant changes to the code, allowing us to reload a boundary geometry during the run, required a new reload function to be written which read the geometry from the DSM and then reinitialized all local variables associated with the body and ensured that the changed



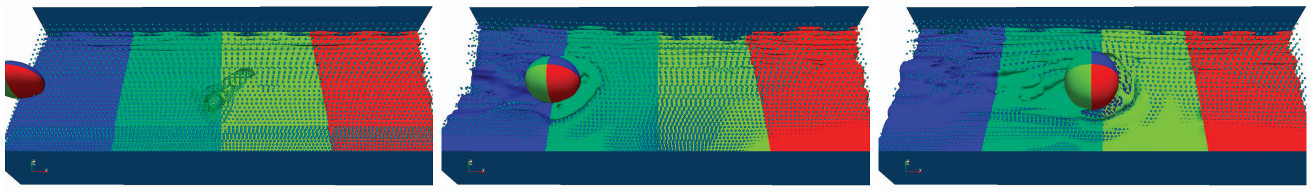
Time: 0.095
Solid Color
Surface With Edges
Spacing 0 0 0
Resolution 64 64 64
Delta 0.0005
Interpolation Method SPH Kernel
Kernel type Spline-3rd-Order
Kernel dimension 3D
H Coefficient (LengthRatio) 1.5
Default Density 1000
Default particle (cube) side length 0.05
Mode: Sequence Time: 0.095 Start Time: 0 End Time: 0.25 No. Frames: 10
Time 0.000e+00 1.250e-01 2.500e-01
Transform1 - Transform - Translate (0) -2 2
Transform1 - Transform - Scale (1) 0.5 1.5
DSM/Disk paused
Source object Transform1
Write full dataset /Mesh_DataSet
Geometry/Points /Mesh_Nodes
Topology/Connectivity /Mesh_Nodes
Field Array /hdf/data/path
Maximum Neighbour Count 128
Kernel Type Spline 3rd Order

Fig. 6. The interface generated for SPH-flow using a template to describe four grids. 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 a box widget transform through the domain and thereby drive the sphere during the simulation. The GUI is setup with resampling and contouring filters (shown in left panels) which are generating the pressure isosurfaces in parallel on four processes (colored). The sphere is also written to the DSM in parallel from the same four processes.

conditions inside the code did not cause failure. This required detailed knowledge of the simulation code and all its internal structures. The more complex the steering intrusion to the simulation then the more difficult the integration—while it is simple to add a new command or parameter to the GUI via the XML—it is much more difficult to ensure that the simulation does what the user wishes of it.

An important result of the integration of the DSM into the SPH-Flow code is shown in Fig. 8 where the total (unoptimized) I/O time for disk and DSM write operations over 15 iterations of the simulation is plotted. The somewhat idealized results of I/O given in Fig. 1 are essentially duplicated. The simulation writes data to DSM almost two orders of magnitude faster than it does to disk—a substantial improvement. The graph also shows CPU time which scales acceptably well up to the 1,536 cores tested, but is almost overtaken by I/O time which scales very badly. Using the DSM to postprocess and reduce the overall I/O is quite feasible. Note that the DSM plot used 36 *pvserver* processes spread over six nodes (contrast with the 88 nodes used for Fig. 8), the value of six nodes was chosen as being 10 percent of the 64 nodes (24 cores each) used for the 1,536 SPH-Flow processes and the DSM size was set to 16 GB, as the largest file written for the 1,536 process case was 12 GB.

Fig. 6 shows a screenshot of a SPH-Flow steering session. The simulation is intended to model a test case of a wedge falling into a fluid, but we are able to stop the simulation and place an arbitrary geometry in instead of the wedge, using a reload command and geometry generated in ParaView. For illustration, we show a sphere generated on four processes written in parallel into the DSM, read in parallel by SPH-Flow and used to compute the flow—which is then exported by SPH-Flow and read by ParaView as a particle data set. Using custom SPH interpolation/resampling filters within ParaView, we compute in parallel the pressure isosurfaces and display them alongside the particles. The sequence of images in Fig. 7 shows snapshots from an accompanying



(a) t_0 (b) t_1 (c) t_2

Fig. 7. Three images from a sequence where a sphere is driven (and deformed) by ParaView and written into the DSM where it is reread by the SPH-Flow fluid simulation. Notice that at t_0 the impression of the wedge object that was present in the fluid when the simulation was initialized is still present, but is replaced by the deviations created by the sphere as time progresses through t_1 and t_2.

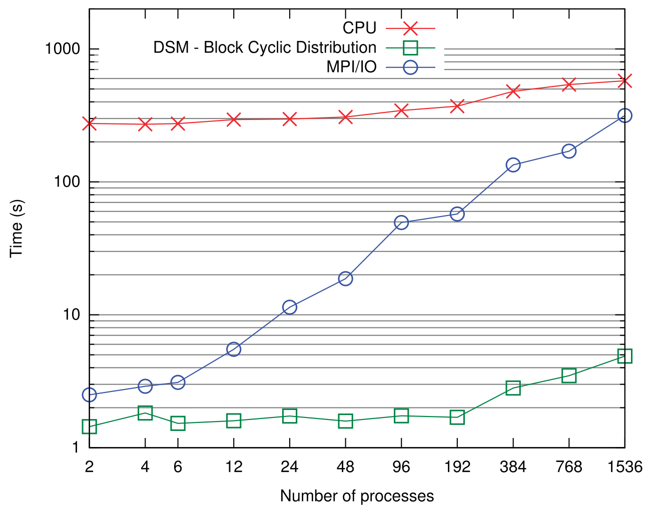


Fig. 8. Comparison of disk-based and DSM (6 nodes, 36 processes, 16 GB) I/O time for the SPH-Flow code. Also shown is the total CPU time during the same run for comparison.

animation where this process is repeated for each time step with the sphere deformed and translated using keyframe animations inside ParaView. This demonstrates quite nicely the ability to couple ParaView pipelines with the simulation and perform live postprocessing and steering. One might argue that operations such as transforming or deforming meshes should be handled inside the simulation rather than outside of it, and in many cases this would be true, but as the complexity of operations required (or imagined), grows, it becomes harder to integrate all features inside the simulation and it makes more sense to couple an application dedicated to these tasks to it.

We have successfully run our steering framework on all 192 cores of our visualization cluster (dynamic MPI connection) and (using static MPI connection of SPH-Flow and ParaView) 1,536 cores on the Cray XE6 (the most cores we can currently successfully run SPH-Flow on). To give the supercomputer a fully interactive feel, we have enhanced the ParaView client server connection dialog to allow us to launch MPMD jobs via the job manager on the Cray directly so that with a single click, we can allocate (subject to node availability) and run simulation + *pvserver* and have them connect back to the ParaView client as if it were a desktop machine.

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 in parallel. 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 arbitrary simulations, or mesh generators and simulations together using a shared virtual file.

The ability to execute analysis pipelines and feed the outputs of those back to the simulation presents a powerful opportunity for optimization loops and what-if scenarios for experimentation. We wish to add Python support to all stages of our ParaView integration so that customized pipelines can be created during normal postprocessing runs and reused during interactive runs in a similar way to the coprocessing framework mentioned earlier. Combining this with the ability to script parameter changes sent back to the simulation will open up a great many new application areas for this environment.

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 the following URLs:

- H5FDdsm: https://hpcforge.org/projects/h5fddsm.
- ICARUS: https://hpcforge.org/projects/icarus.

References

[1] A. Henderson, *ParaView Guide, A Parallel Visualization Application.* Kitware, Inc., http://www.paraview.org, 2005.
[2] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel Computational Steering and Analysis for HPC Applications Using a ParaView Interface and the HDF5 DSM Virtual File Driver," *Proc. Eurographics Symp. Parallel Graphics and Visualization,* T. Kuhlen, R. Pajarola and K. Zhou eds., pp. 91-100, 2011.
[3] J. Brooke, P. Coveney, J. Harting, S. Jha, S. Pickles, R. Pinning, and A. Porter, "Computational Steering in RealityGrid," *Proc. UK E-Science All Hands Meeting,* pp. 885-888, 2003.
[4] N. Richart, A. Esnard, and O. Coulaud, "Toward a Computational Steering Environment for Legacy Coupled Simulations," *Proc. Sixth Int'l Symp. Parallel and Distributed Computing (ISPDC),* p. 43, July 2007.
[5] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max, "A Contract Based System for Large Data Visualization," *Proc. IEEE Visualization (VIS '05),* pp. 191-198, Oct. 2005.
[6] B. Whitlock, J.M. Favre, and J.S. Meredith, "Parallel in Situ Coupling of Simulation with a Fully Featured Visualization System," *Proc. Eurographics Symp. Parallel Graphics and Visualization,* T. Kuhlen, R. Pajarola and K. Zhou eds., pp. 101-109, 2011.
[7] N. Fabian, K. Moreland, D. Thompson, A.C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K.E. Jansen, "The ParaView Coprocessing Library: A Scalable, General Purpose in Situ Visualization Library," *Proc. IEEE Symp. Large-Scale Data Analysis and Visualization (LDAV),* Oct. 2011.
[8] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," *Proc. 19th ACM Int'l Symp. High Performance Distributed Computing (HPDC),* pp. 25-36, 2010.
[9] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, Metadata Rich IO Methods for Portable High Performance IO," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS),* pp. 1-10, 2009.
[10] F. Zhang, C. Docan, M. Parashar, and S. Klasky, "Enabling Multi-Physics Coupled Simulations within the PGAS Programming Framework," *Proc. IEEE/ACM 11th Int'l Symp. Cluster, Cloud and Grid Computing (CCGrid),* pp. 84-93, 2011.
[11] V. Vishwanath, M. Hereld, V. Morozov, and M.E. Papka, "Topology-Aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC),* pp. 19:1-19:11, 2011.
[12] J. Soumagne, J. Biddiscombe, and J. Clarke, "An HDF5 MPI Virtual File Driver for Parallel in-Situ Post-Processing," *Proc. 17th European MPI Users' Group Meeting Conf. Recent Advances in the Message Passing Interface,* R. Keller, E. Gabriel, M. Resch, and J. Dongarra eds., pp. 62-71, 2010.

[13] J. Soumagne and J. Biddiscombe, "Computational Steering and Parallel Online Monitoring Using RMA through the HDF5 DSM Virtual File Driver," *Proc. Int'l Conf. Computational Science (ICCS),* vol. 4, pp. 479-488, 2011.

[14] W. Gropp and R. Thakur, "Revealing the Performance of MPI RMA Implementations," *Proc. 14th European PVM/MPI Users' Group Meeting Recent Advances in Parallel Virtual Machine and Message Passing Interface,* F. Cappello T. Herault and J. Dongarra eds., pp. 272-280, 2007.

[15] J. Soumagne, J. Biddiscombe, and A. Esnard, "Data Redistribution Using One-Sided Transfers to in-Memory HDF5 Files," *Proc. 18th European MPI Users' Group Meeting Conf. Recent Advances in the Message Passing Interface,* A. Danalis, D. Nikolopoulos and J. Dongarra eds., pp. 198-207, 2011.

[16] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre, "Remote Large Data Visualization in the ParaView Framework," *Proc. Eurographics Symp. Parallel Graphics and Visualization,* B. Raffin, A. Heirich, and L.P. Santos eds., pp. 163-170, 2006.

[17] J.A. Clarke and E.R. Mark, "Enhancements to the eXtensible Data Model and Format (XDMF)," *Proc. DoD High Performance Computing Modernization Program Users Group Conf. (HPCMP-UGC '07),* pp. 322-327, 2007.

[18] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, "Interactive SPH Simulation and Rendering on the GPU," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA '10),* pp. 55-64, 2010.

[19] G. Oger, E. Jacquin, P.-M. Guilcher, L. Brosset, J.-B. Deuff, D.L. Touze, and B. Alessandrini, "Simulations of Complex Hydro-Elastic Problems Using the Parallel Sph Model SPH-Flow," *Proc. Fourth SPHERIC Int'l Worksop,* 2009.

John Biddiscombe received the BEng degree in electronic engineering from Warwick University in 1989 and worked at the Rutherford Appleton Laboratory on Digital Signal Processing methods for altimetry and remote-sensing radar. Following this he implemented 3D algorithms for cellular radio propagation, in particular, modeling signals around buildings and trees and designed a propagation toolkit derived from the VTK library for these simulations. This work in turn led to an interest in visualization algorithms and he now works (since 2004) as a HPC/Visualization Scientist at the Swiss National Supercomputing Centre with a specialization in parallel visualization and supercomputing.

Jerome Soumagne received the MSc/engineer diploma in computer science in 2008 from ENSEIRB Bordeaux. He registered at INRIA Bordeaux and Bordeaux University to follow doctoral studies, he joined the Swiss National Supercomputing Centre as a PhD student in the Scientific Computing Research group to work toward the thesis, while working on the NextMuSE European project. His main research interests are parallel coupling and data transfer between HPC applications, and parallel visualization. He is a student member of the IEEE.

Guillaume Oger received the engineer diploma from the Ecole Centrale de Nantes in 2003, and the PhD degree from the University of Nantes/Ecole Centrale de Nantes in 2006, in fluid mechanics and HPC, and more especially on the Smoothed Particle Hydrodynamics (SPH) method. He worked during three years (2007-2010) as a researcher/developer at HydrO-Ocean, a company specialized in the creation of innovative software for complex fluid simulations for the industry. His research interest mainly concerns numerical modeling of high-dynamic fluid flows, with a focus on distributed and shared memory parallelization. He is currently working in the Fluid Mechanics Laboratory of the Ecole Centrale de Nantes as a research engineer.

David Guibert received the engineer diploma from ISTIL in 2004 and the PhD degree in 2009 from the University of Lyon, in applied mathematics and HPC. He worked with several companies to enhance their "in-house" code. His research interests include numerical methods and suitable schemes for HPC platforms, new advances in computer science that ease the development of algorithms to perform highly complex computations.

Jean-Guillaume Piccinali received the graduate degree from the National School of Meteorology, Toulouse, in 2001. He works as a High Performance Computing application analyst at the Swiss National Supercomputing Centre (CSCS). Before joining CSCS in 2008, he worked at IBM, BULL, and CNRS as a High Performance Computing specialist and has been involved in Scientific Computing since 2001. His current responsibilities involve parallel programming, code optimization, user support, and training courses.