N° d'ordre: 4643

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

Par Jérôme SOUMAGNE

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ: INFORMATIQUE

An In-situ Visualization Approach for Parallel Coupling and Steering of Simulations through Distributed Shared Memory Files

---

Soutenue le : 14 décembre 2012

Après avis de :

M. Gabriel	**ANTONIU**	Research Director, INRIA Rennes
M. Witold	**DZWINEL**	Professor, AGH UST Krakow

Devant la commission d'examen formée de :

M. Gabriel	**ANTONIU**	Research Director, INRIA Rennes	Président & Rapporteur
M. John	**BIDDISCOMBE**	Visualization Scientist, CSCS Lugano	Co-directeur de Thèse
M. Olivier	**COULAUD**	Research Director, INRIA Bordeaux	Examinateur
M. Witold	**DZWINEL**	Professor, AGH UST Krakow	Rapporteur
M. Aurélien	**ESNARD**	Assistant Professor, University of Bordeaux	Co-directeur de Thèse
M. Jean	**ROMAN**	Professor, INRIA & Universiy of Bordeaux	Directeur de Thèse

Abstract

As simulation codes become more powerful and more interactive, it is increasingly desirable to monitor a simulation *in-situ*, performing not only visualization but also analysis of the incoming data as it is generated. Monitoring or post-processing simulation data *in-situ* has obvious advantage over the conventional approach of saving to—and reloading data from—the file system; the time and space it takes to write and then read the data from disk is a significant bottleneck for both the simulation and subsequent post-processing steps. Furthermore, the simulation may be stopped, modified, or potentially steered, thus conserving CPU resources.

We present in this thesis a loosely coupled approach that enables a simulation to transfer data to a visualization server via the use of *in-memory* files. We show in this study how the interface, implemented on top of a widely used hierarchical data format (HDF5), allows us to efficiently decrease the I/O bottleneck by using efficient communication and data mapping strategies. For steering, we present an interface that allows not only simple parameter changes but also complete re-meshing of grids or operations involving regeneration of field values over the entire computational domain to be carried out. This approach, tested and validated on two industrial test cases, is generic enough so that no particular knowledge of the underlying model is required.

Keywords—Parallel I/O, Distributed Shared Memory, Communication Models, Computational Modeling, Data Models, Synchronization.

Résumé

Les codes de simulation devenant plus performants et plus interactifs, il est important de suivre l'avancement d'une simulation *in-situ*, en réalisant non seulement la visualisation mais aussi l'analyse des données en même temps qu'elles sont générées. Suivre l'avancement ou réaliser le post-traitement des données de simulation *in-situ* présente un avantage évident par rapport à l'approche conventionnelle consistant à sauvegarder—et à recharger—à partir d'un système de fichiers; le temps et l'espace pris pour écrire et ensuite lire les données à partir du disque est un goulet d'étranglement significatif pour la simulation et les étapes consécutives de post-traitement. Par ailleurs, la simulation peut être arrêtée, modifiée, ou potentiellement pilotée, conservant ainsi les ressources CPU.

Nous présentons dans cette thèse une approche de couplage faible qui permet à une simulation de transférer des données vers un serveur de visualisation via l'utilisation de fichiers *en mémoire*. Nous montrons dans cette étude comment l'interface, implémentée au-dessus d'un format hiérarchique de données (HDF5), nous permet de réduire efficacement le goulet d'étranglement introduit par les I/Os en utilisant des stratégies efficaces de communication et de configuration des données. Pour le pilotage, nous présentons une interface qui permet non seulement la modification de simples paramètres, mais également le remaillage complet de grilles ou des opérations impliquant la régénération de grandeurs numériques sur le domaine entier de calcul d'être effectués. Cette approche, testée et validée sur deux cas-tests industriels, est suffisamment générique pour qu'aucune connaissance particulière du modèle de données sous-jacent ne soit requise.

Mots-clefs—I/Os Parallèles, Mémoire Partagée Distribuée, Modèles de Communication, Modélisation Informatique, Modèles de Données, Synchronisation.

Acknowledgments

This thesis work was supported by the NextMuSE project receiving funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under grant agreement 225967. It has been realized in collaboration between the Swiss National Supercomputing Centre (CSCS) and the HiePACS[1] project-team, which is a joint project of Inria, University of Bordeaux and CNRS (LaBRI UMR 5800).

Of the many people who deserve thanks, some are particularly prominent. I would first like to thank my thesis co-supervisor, John Biddiscombe, who gave me the possibility to realize this work at CSCS within the NextMuSE European project. I thank him for all the advice and directions he has been giving to me and for the excellent working environment provided during these three years. I would also like to thank my thesis supervisor, Jean Roman, and Aurélien Esnard, my second co-supervisor, for all the guidance, corrections and reading work of this thesis. Thanks also to all the CSCS staff and particularly Neil Stringfellow, Jean-Guillaume Piccinali, Nina Suvanphim (Cray) and Sadaf Alam for the help and directions in obtaining some of the results that are presented in this thesis. Thanks also to Thomas Schulthess and Michele de Lorenzi who extended my contract and let me have an extra time to finalize this work.

I would also like to thank my CSCS neighbors and former neighbors from *via al Ponte*, Jeff and Tim, for all the help in improving my English and for the good laughs.

Finally I apologize for anyone I forgot to mention in these acknowledgments and who contributed to this work.

Jérôme Soumagne

[1] `http://hiepacs.bordeaux.inria.fr`

Contents

List of Figures

List of Tables

"Computational science and engineering encompass a broad range of applications with one common denominator: visualization."

— Bruce H. McCormick

Chapter 1.

Introduction

In science and engineering, mathematical modeling aims at creating abstract representations for the understanding and prediction of physical phenomena. For observation of the phenomenon or validation of the defined model an experimentation phase can generally be defined; however, in most cases, experimentation is expensive and not always feasible (e.g., galaxy collision). In addition to this, modeling a system implies solving complex equations, for which in several cases no analytic solutions can be found. In this context, numerical simulation can allow one to find approximate solutions and understand a phenomenon under controlled conditions.

1.1. Numerical Simulation and Scientific Visualization

A *numerical simulation* program consists of solving a system numerically by iterating over time or space (depending on what needs to be solved) until the system reaches a convergence point. At each iteration step, a data output that contains a representation of the physical state of the system may be produced and further post-processed for analysis of the simulation results. Note that as the accuracy of the solution produced is an important factor, one may need to increase the model resolution to get more accurate results, although this may in turn lead the simulation to consume more resources. For example in figure 1.1a, in a mesh-based representation, increasing the mesh resolution (and hopefully the accuracy of the representation) requires more resources in terms of space to store the data and more resources in terms of computing power to be able to process a larger amount of data in a still reasonable amount of time.

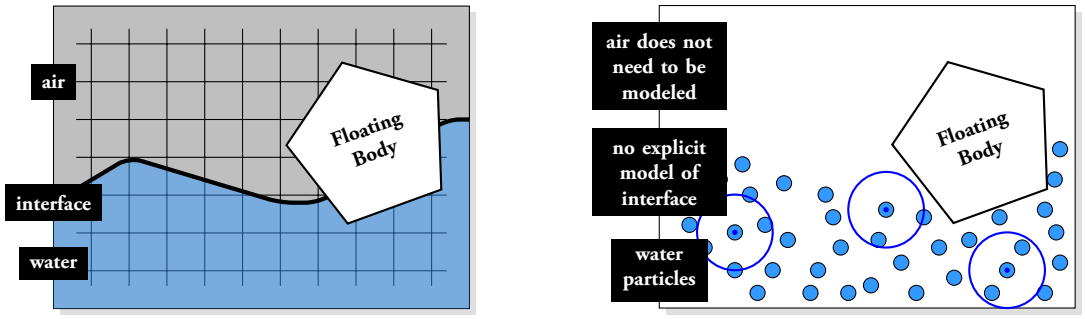
As the amount of data that is to be produced by the simulation is generally too large to be humanly readable, data must be post-processed before one can conclude anything about the results. This step, called *scientific visualization*, allows scientists to get a relevant representation of the data and highlight specific data regions or variables (e.g., velocity, pressure, etc). While this process

has been used for decades, it requires definition of a common and well-suited interface between simulation and visualization applications.

In addition to this process, it may also be useful for an engineer or a scientist to directly interact with the simulation code while visualizing data and change parameters and variables and/or data objects, and see the corresponding effects and evolution as the simulation carries on computation. This process, called *computational steering*, is used in several applications and can allow one to recreate a fully simulated environment by modifying the right parameters and hence the simulation dynamics during computation (removing the need to relaunch computation from modified initial conditions).

1.2. Multidisciplinary Environment and ICARUS Approach

Different approaches can be used to simulate a single phenomenon and data models of simulation codes can therefore be structured in various ways, depending on the representation that is to be defined. For example, in CFD (Computational Fluid Dynamics) one can simulate a fluid using a standard mesh-based method. However because the interface between air, water, and bodies is variable and needs to be modeled (which can be very complex), mesh-free particle methods like SPH (Smooth Particle Hydrodynamics [51]) have emerged as an alternative to classical mesh-based methods. As illustrated in figure 1.1, instead of modeling the material with a fixed spatial mesh, the fluid (and the solid) can be modeled with moving particles. Particles have no explicit connectivity, but interact with neighbors within a specified range. This approach brings advantages in the treatment of complex physics with interfaces for moving bodies.



(a) Classical mesh-based approach. **(b)** Mesh-free particle approach.

Figure 1.1. Different approaches to model a body floating in water.

As an illustrative example that we will use in this thesis (see chapter 5), the NextMuSE [24] European project defines a multidisciplinary environment between different simulation codes, which

may use different data representations, in various domains such as energy, transport and healthcare. The objective of NextMuSE is to initiate a paradigm shift in Computational Fluid Dynamics (CFD) and Computational Multi-Mechanics (CMM) simulation software, which is used to model physical processes. Relying on SPH methods (fundamentally different from mesh-based techniques as explained in figure 1.1), the project offers the possibility of a novel and adaptive framework for user interaction, and has the potential for integrated multi-mechanics modeling in applications where traditional methods fail.

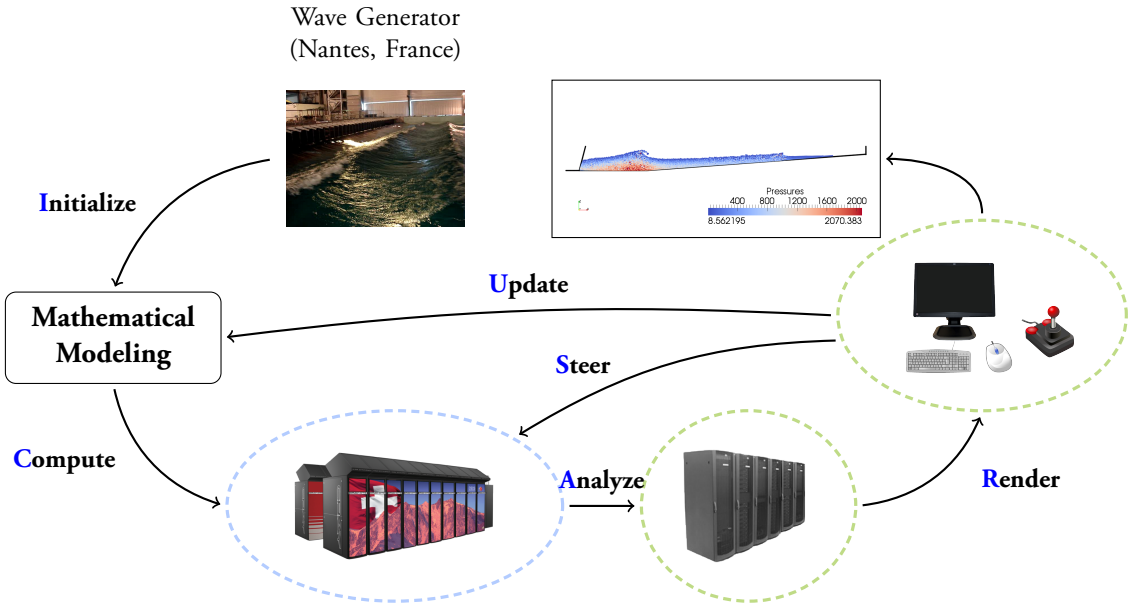


Figure 1.2. NextMuSE problematic and ICARUS concept.

In this context, we define objectives and research axes to create a framework and platform, capable of interfacing multiple simulation codes that use different data models to a visualization and analysis application at a minimal cost for the user. We define the concept of ICARUS—Initialize Compute Analyze Render Update Steer [24]—which is illustrated in figure 1.2.

Based on experimental observations, one may define a mathematical model. From this model, a simulation code is implemented and the computation occurs on a large *high-performance computing* (HPC) machine. This composes the simulation part, represented in blue[1]. Depending on the use case, data output may be sent to other resources (machine or set of nodes) for analysis, which may even be connected to a visualization client. This is represented in green[1] and must be generic enough to be able to support the different simulation codes and data models. The visualization client may produce images but a user must also be able to *steer* the simulation by dynamically modifying parameters or meshes/data, so that he can explore and understand the effects of the different

[1] color used in the different figures of this discussion.

component modifications onto the simulation. This may in turn lead the simulation developer to *update* the mathematical model, and improve the correctness or accuracy of the simulation.

1.3. Objectives

To follow the ICARUS concept, the first objective of this work is to create an interface that allows one to re-route simulation data output to a post-processing server for analysis and visualization without any significant rework of the simulation codes. As described above, the interface must be generic enough to be used with any data model. The second objective is to define a comprehensive steering interface so that not only parameters or simple scalars can be exchanged but also complete meshes and data objects.

To respond to these two main objectives, it is necessary to define an interface that is flexible enough to allow two-way communications between simulation and post-processing applications and generic enough to allow transfers of various data structures. As we want also to be able to run on large HPC machines, the interface needs to be able to scale up to thousands of processors and therefore avoid overhead wherever possible, neither on the simulation side nor on the post-processing side; hence communications must also execute as fast as possible.

1.4. Outline

In chapter 2 we present the existing architectures, interfaces and means of exchange between simulation and post-processing applications that can respond to our problem and see how we can position ourselves to determine the best suitable architecture that can ensure an efficient interfacing of the different simulation codes with post-processing applications. In chapter 3, we explain the different architectural choices that we make[2], which sub-problems this leads us to, which solutions or compromises we find and more importantly, which key concepts our architecture needs to achieve the best performance. We present our implementation in chapter 4 and validate the different concepts introduced and used in our architecture through unit test cases. In chapter 5, we apply this work to two different test cases from the NextMuSE project and validate our approach as well as our architectural and implementation choices. Finally, we conclude on the work that has been achieved and present how the different problems encountered can lead in the future to new improvements and implementation solutions.

[2]Note that most of the work that we will present in the next chapters has been published in [80–84].

Chapter 2.

In-situ Visualization and Steering Approaches

Consuming more and more resources, simulations also generate larger and larger amounts of data. The significant bottleneck introduced by the analysis or the visualization of the data produced requires digging for new post-processing methods and techniques. In this chapter we show how the limitations of the original model, referred to as *traditional model*, lead to new approaches to reduce the amount of data that is to be processed by a given post-processing element. While choosing a technique may restrict the operations that can be applied to the data, the degrees of freedom for steering the simulation may vary accordingly.

2.1. From Traditional Visualization…

As illustrated in figure 2.1, in a traditional approach, a simulation writes data sequentially or in parallel to disk using a defined file format. A post-processing or visualization application is then used to read from the file system and visualize the generated data. This approach, used by several tools, presents the advantage of giving to the user a generic solution to post-process his data and, as simulation and post-processing are entirely decoupled, it does not interfere in any way with the simulation, potentially still running.

Several visualization applications such as ParaView [15, 70] or VisIt [19], based on the VTK library [39], make use of this approach. Filters are applied to a source object (in this case, the data read from the file system), which create from the original raw data a new set of data that is to be visualized. This new set of data is then converted into geometric objects (mappers), which are then rendered and displayed onto a screen. One can therefore create a complex post-processing

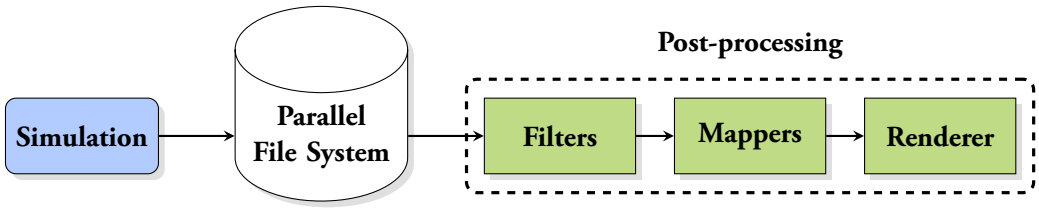


Figure 2.1. Data is read from a parallel file system, the analysis and visualization steps are performed using a post-processing pipeline composed of filters, mappers and renderer.

pipeline, composed of several filters and mappers, the complexity of the data and the number of operations requested increasing the time needed by the analysis to complete.

2.1.1. Parallel File Interfaces and Data Formats

Parallel file interfaces are now used in several applications and achieve a reasonable good performance on well-known parallel file systems such as GPFS [38], Lustre [59], etc. Among the different parallel interfaces available, some have distinguished themselves for their flexibility, performance, and reliability.

MPI I/O

MPI I/O, defined in the MPI-2 standard [2, 33], is a parallel I/O interface. Implementations, ROMIO [71] or more recently OMPIO [16], have been designed on top of abstract I/O device layers that enable portability to underlying I/O systems. One of the most important implemented features is collective I/O operations, which can even be non-blocking now. Collective I/O operations adopt a two-phase I/O strategy and improve the parallel I/O performance by significantly reducing the number of I/O requests that would otherwise result in many small, non-contiguous I/O requests. However, MPI I/O reads and writes data in a raw format without providing any functionality to effectively manage the associated metadata (see below), and thus does not guarantee data portability. It is therefore not the most convenient file format for scientists who need to organize, transfer, or share their application data.

HDF5

HDF5 [72] is a widely used portable file format and library developed by the HDF Group for storing, retrieving, analyzing, visualizing (assuming an appropriate reader is provided) and converting data. HDF5 stores multidimensional arrays along with metadata in a file. It supports hierarchical

file structures providing users with a high degree of flexibility for data management. As illustrated in figure 2.2, a dataset is mapped onto a file and its memory description and layout is stored in metadata. This allows users to define and organize their data into different groups and datasets and build an hierarchical tree of data, all the file mapping being taken care of by the library.

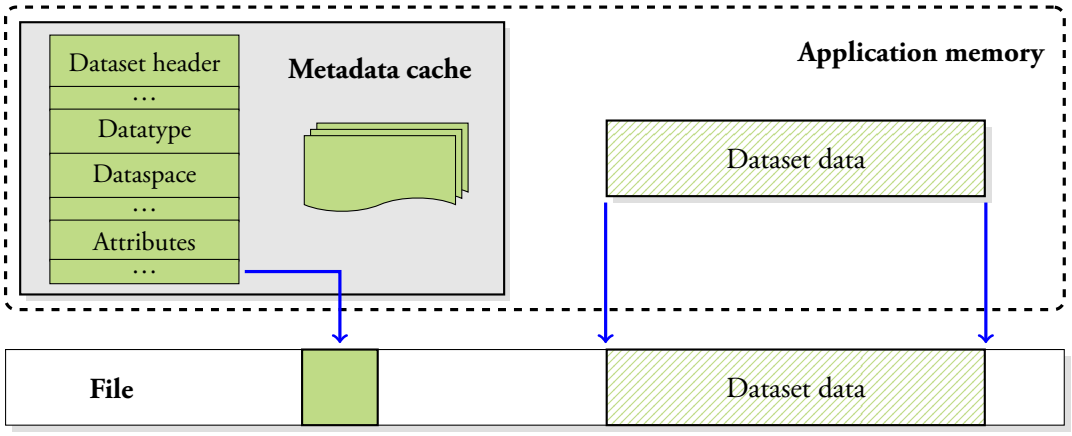


Figure 2.2. HDF5 stores multidimensional arrays (datasets) along with metadata in a file. In this case, a dataset is contiguously mapped from the application memory to a file, its data structure being described in the metadata cache.

HDF5 supports parallel data access built on top of MPI I/O (which is best suited for parallel file systems as previously described). HDF5 also provides its own ways of tuning parallel data writes. For instance, the chunking mechanism allows files and particularly datasets to be stored in a non-contiguous form (i.e., in equally sized chunks); this can be helpful for parallel file systems, over which datasets can be striped. Additional optimization has also been made in the HDF5 library for specific file systems, such as the Lustre file system [37], and features such as variable-size arrays and data compression are made possible by partitioning the storage space into chunks.

NetCDF

The Network Common Data Form (NetCDF) [63] format is another portable file format and programming interface used in the scientific community (especially atmospheric science community) for data access and storage of structured datasets. NetCDF uses a linear data layout in which data arrays are contiguous or interleaved in a regular pattern. Parallel NetCDF (PnetCDF) [42] is a parallel version of NetCDF developed by Argonne National Laboratory and Northwestern University and is built on top of MPI I/O to provide efficient parallel file accesses through the use of collective I/Os. One of the goals of NetCDF is to support efficient access to small subsets of large datasets. To support this goal, NetCDF uses direct access rather than sequential access. This can be much more efficient when the order in which data is read is different from the order in which it

was written, or when it must be read in different orders for different applications. NetCDF4 [64] uses HDF5 as a data storage layer, HDF supports n-dimensional datasets and each element in the dataset may itself be a complex object. Therefore the use of HDF5 as a data format adds a significant overhead in metadata operations as creation of multiple objects implies multiple metadata accesses. NetCDF does not support compression directly but allows users to use the HDF5 interface for data compression.

2.1.2. Limitations of the Traditional Approach

As previously mentioned, simulation codes become nowadays more complex, use more and more resources, and produce larger and larger data. Even with smart file formats, the conventional approach of saving to—and reloading data from—the file system now shows its limitations. This is illustrated by table 2.1, which shows an historical overview of different existing large systems [29, 47, 55, 68] such as Cray XT5 systems, now decommissioned systems such as ASCI machines [67], and upcoming systems such as the OLCF Cray XK6 [56] or ASC IBM BG/Q [41]. It is clear from this table that, as the peak performance grows, the gap between the amount of memory available on the system and the file system bandwidth increases. Therefore, assuming that a simulation runs on the full system, the time required to perform a whole system checkpoint to disk increases as well. A few exceptions in this table, such as the RIKEN K computer or Roadrunner, show lower checkpoint times, bringing a faster file system compared to the others for the same period. However the overall trend, even for these systems, is not to get a smaller ratio (*System Memory*) / (*File System Bandwidth*).

Table 2.1. Historical peak performance of supercomputers and associated I/O rates.

Machine	Year	Peak (TFlops)	System Memory (TB)	File-System BW (GB/s)	Whole System Checkpoint (s)
ASCI Red	1997	1.8	1.1	4	≃ 280
ASCI White	2001	12	6	12	≃ 510
ASC Red Storm	2005	41	33	50	≃ 680
Roadrunner	2008	1344	104	216	≃ 490
CSCS XT5	2009	212	29	16	≃ 1860
NCCS XT5	2009	2332	292	240	≃ 1250
RIKEN K Computer	2011	11550	1377	≃ 1024	≃ 1380
OLCF XK6	201x	≃ 15360	584	≃ 600	≃ 1000
ASC Sequoia	201x	≃ 20480	1638	≃ 650	≃ 2580

As a direct consequence of these I/O limitations, post-processing applications suffer heavily, as they pay twice the accesses to the file system: one is introduced by the simulation writing to disk and the other by the post-processing application reading from it (and the bandwidth from the compute nodes to the file system may be substantially better than from the file system to the post-processing nodes).

Besides this, computational steering, which can also be applied in a traditional approach by visualizing and analyzing every time step as it is written to disk, may suffer from interaction issues as the frequency of data outputs diminishes. For example, InSt [48] is a framework that allows one to steer a simulation remotely. First the simulation output is written to the file system and a service sends it via the network to a remote machine. Visualization is then performed and a steering action may be sent back. Whereas this approach is useful when a simulation can only run on a specific site for security reasons, if the amount of data produced by the simulation drastically increases, the interaction and data exploration capabilities brought by the steering approach will be considerably reduced.

Still, the previous statement of this section does not mean that codes should not produce outputs to disk any more, as one may need to save his data for a future usage and data post-processing may only be feasible from the original raw data; but more specifically, this means that in a regular use, when data archiving is not necessary, other solutions must be sought to minimize the usage of disks.

2.1.3. Solutions to Minimize Cost of I/Os

Several solutions exist to minimize the costs of I/Os[1]. Some of these solutions are presented in [18] where post-processing systems have to meet petascale simulation systems requirements and they are already used by many applications such as ParaView, VisIt, etc. We present in the following points three of these techniques, which follow a *traditional* post-processing model.

Pure Parallelism

The first natural technique to minimize I/O costs is to have several processes reading in parallel (see figure 2.3), so that each of them can work on its own subset.

This technique is still one of the most used techniques but presents some limitations. Assuming that a significant number of nodes are used to access the file-system, the I/O bottleneck may be

[1] Note that I/Os is here an abuse of terminology and specifically refers (unless specified) in most of this document to "disk I/Os".

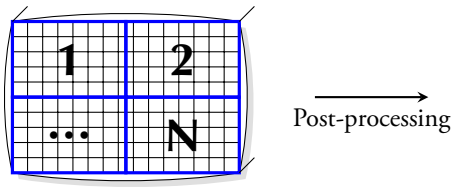


Figure 2.3. Data pieces are read and treated in parallel.

proportionally reduced. However, the main issue one may need to consider is that the size of the cluster used to post-process data is usually not as large as the one used for the simulation. While the simulation may minimize the I/O costs by writing in parallel, the post-processing machine, when reading in parallel, will have to pay at a larger extent the size of the data. This technique presents therefore an important limitation, as the simulation may scale to a high number of processes, but post-processing may still be limited by parallel I/Os from disk.

Multiresolution

The second idea one may have is to work on a simplified version of the data for post-processing by temporarily decreasing the resolution (see figure 2.4). When the coarse representation is no longer needed, one may switch back to the full resolution representation and run the analysis again if necessary.

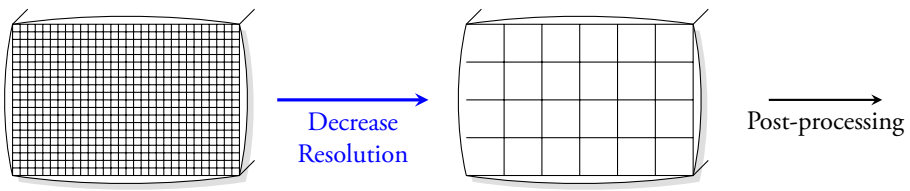


Figure 2.4. Data resolution is temporarily decreased for post-processing.

For example in [43], the authors construct a multiresolution hierarchy based on subdivisions and make use of downsampling filters for high quality data approximation on each level of detail. Whereas this can be seen as a good alternative (particularly for data exploration), the main drawback of this technique is the consequence of operating on a simplified version of data. It may not be always meaningful as one may miss details or get different results from the analysis and therefore may need to go back to higher resolutions.

Out-of-core Processing

Subsetting or out-of-core processing [69] consists of operating on a unique partitioned data subset, fitting in the main memory. Each subset is then processed, one at a time (see figure 2.5).

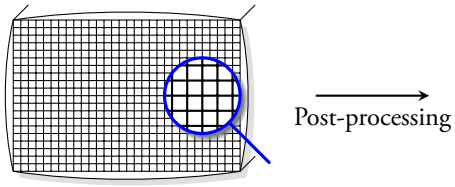


Figure 2.5. Post-processing occurs serially onto partitioned data subsets.

This method obviously reduces the accesses to disk as only small blocks of data are read. Additionally post-processing application components can be multithreaded or distributed over different nodes and in this context, task parallelism can allow the current subset to be post-processed while the next subset is being read.

One issue remains: depending on the data that is to be analyzed/visualized, partitioning the data so that only a very small subset of data is used and analyzed may not always be applicable, and I/O costs may still be high as depending on the size of data, a significant number of I/O operation will still be necessary.

2.2. …To In-situ Visualization

As data becomes ever larger [66], techniques presented above do not allow sufficient reduction of I/O costs. To decrease the I/O bottleneck even further, the concept of *in-situ* visualization or *in-situ* processing is re-introduced[2]. Generally speaking, *in-situ* visualization of a simulation is a technique where a simulation is coupled with a visualization interface and visualization (or post-processing) occurs while the simulation is running. The expression "*in-situ* processing" can be ambiguous as several methods use the same terminology for different modes of operation. We will separate in the next sections *in-transit* processing [52] (also commonly called *staged* processing)—where data movement between different nodes occurs—from *in-position* processing—where no data movement occurs and memory spaces are shared between simulation and post-processing applications.

[2]Not completely new [50], even though a lot of interest has been shown for this technique in the last few years and particularly because of the increase of I/O constraints.

More generally, *in-position* processing is a synchronous approach (i.e., a tightly coupled approach) and *in-transit* is asynchronous (i.e., a loosely coupled approach). As we will see further in details, each of these approaches has its own advantages and drawbacks. As these different approaches imply a different level of coupling, we also present what this implies in terms of steering capabilities that can be associated to the simulation.

2.2.1. Tightly Coupled Approach

In a tightly coupled approach, as illustrated in figure 2.6, simulation and post-processing share the same memory. A post-processing application that uses this method can therefore have direct access to the requested memory regions without having to do any additional memory copies and of course, any data transfer with the simulation (i.e., data movement costs are close to zero).

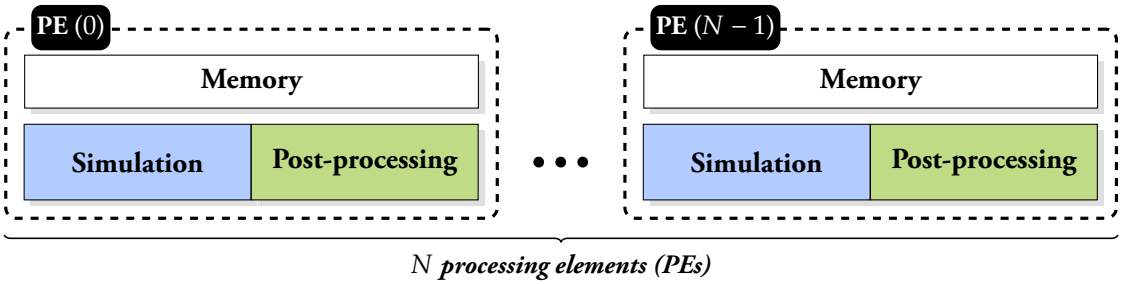


Figure 2.6. Simulation and post-processing applications share the same memory.

For instance, VisIt [19] provides users with the libsim [76] in-situ visualization library, a lightweight library that is portable enough to be executed on a large variety of HPC systems. The library defines an API so that one can simply interface to the VisIt environment. This implies for the user annotation of the code with the required functions. The VisIt libsim also provides a way of dynamically connecting to a simulation already running so that one can monitor results, removing the need to launch both applications at the same time. This is ensured by periodically calling `VisitDetectInput` from the simulation main loop (see figure 2.7) so that when requested, new connections between the VisIt client and the libsim library are created.

On the other hand, ParaView proposes a similar approach with the coprocessing library [28]. At the end of a simulation time step, the simulation makes a function call to pass the current simulation/solution state to the ParaView coprocessor. The coprocessor reads then instructions from a Python script to build a filter pipeline for analysis of the data. The post-processed data, which may even be at the end of this process a simple image, can be directly saved using disk I/Os.

Both libraries require some re-working of the code so that memory addresses can be passed to the interface; 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. In most cases, post-processing operations have to be well defined before running the simulation, which means that only a specific sequence of events can be called during a post-processing request.

As described in [77] where a full in-situ visualization pipeline is applied to combustion simulations at large scale, making use of these approaches for in-situ visualization means that the analysis will run on the same computing cores as the simulation, placing additional memory demands on them. It is also likely that as the simulation algorithm scales up to a high number of cores, the analysis algorithms that need to be applied to the simulation output do not scale as well, leading to either additional communication overheads or computation bottlenecks.

Consequently, since post-processing and simulation access the same memory, post-processing must operate synchronously with the simulation (and this can be a potential drawback as while the analysis is being processed, the simulation must *wait*). In other terms, as illustrated by figure 2.7, a new simulation time step can be computed only when the local post-processing of the current time step is completed (if post-processing was requested).

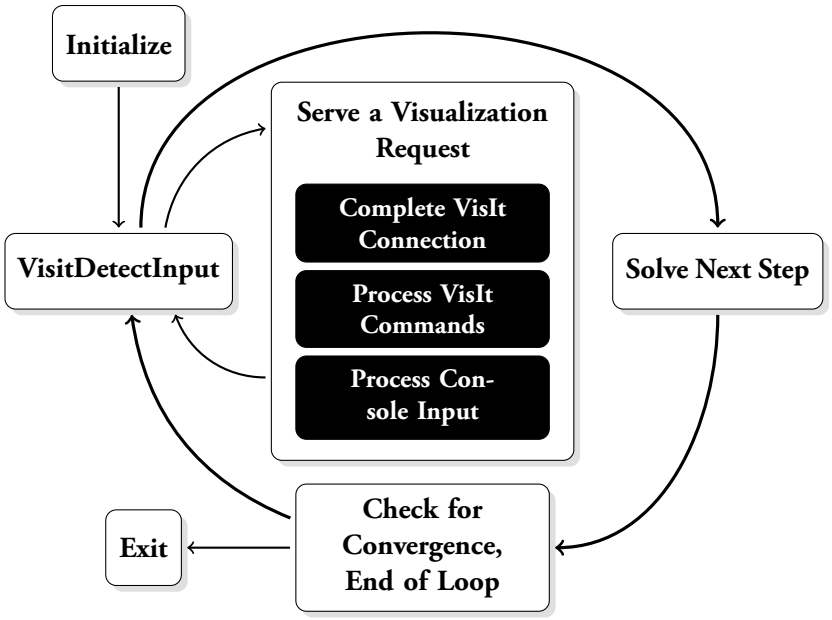


Figure 2.7. Simulation control flow after introducing in-situ processing with VisIt libsim.

While the restrictions cited above affect the overall computation time, they also reduce the degrees of freedom one may require to steer a simulation. Memory constraints have a high impact

and one may need to pass back to the simulation different memory objects or even new objects; depending on the use case, this may be very complex to handle with this approach. Furthermore, because simulation and post-processing are *tightly* coupled (and pipeline is pre-defined), the level of interactivity one may require to explore the data—and add potential analysis and specific steering actions—may be more limited or more complex to handle.

2.2.2. Loosely Coupled Approach

In a loosely coupled approach, visualization and analysis run concurrently on different resources, as illustrated in figure 2.8. When a new time step is computed, output is sent via the network to another set of nodes (or machine[3]).

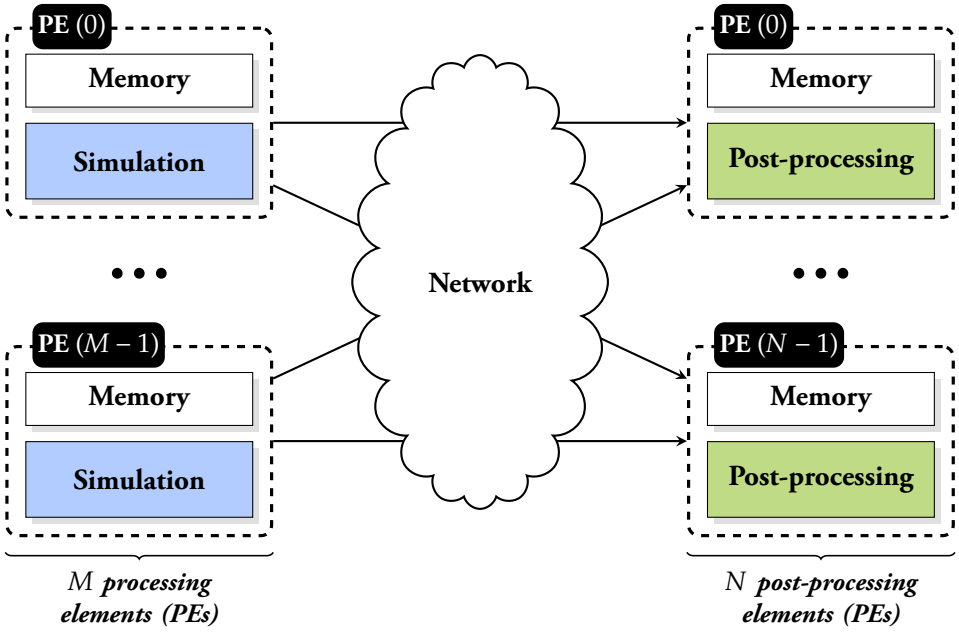


Figure 2.8. Simulation and post-processing run on separate physical nodes and data is transferred from one application to the other through the network.

A first example of tools that uses this approach is the EPSN library [65], which defines a parallel high level loosely coupled model by manipulating and transferring distributed objects such as parameters, grids, meshes and points between different applications across the network, using a CORBA [57] communication protocol. A user can ask for objects and these objects are automatically mapped (and redistributed) using the EPSN model to VTK sources (in-memory), or any other output format (e.g., HDF5 for which a module in the library is provided as is a

[3]But in this case, the term of in-situ processing is not really appropriate as the internal network is no longer used.

ParaView plug-in for visualization). EPSN includes a mesh redistribution layer that maps grids from N simulation processes to M post-processing processes. One can then easily interface the simulation or visualization code to one of the mappers, and define steerable parameters and actions. The EPSN library also makes use of XML files to describe the data and interactions and provides *task descriptions* that can be used to define synchronization points at which codes can wait for each other.

Another *in-transit* (and therefore loosely coupled) approach has been introduced in the ADaptable I/O System (ADIOS) framework [44]. ADIOS has been designed to separate the I/O API from the actual implementation of the I/O methods. It implements a new file format called the BP format, which can easily be converted into common formats such as HDF5, NetCDF or ASCII. Using the same implementation and an XML description file, one can switch between the different services that the library provides the user with, and also select specific I/O services that can for example be used to map the data output to a remote memory (with no code changes). Particularly, ADIOS defines the DataSpaces method [26] to create a virtual shared memory space, a staging area that can be asynchronously accessed using one-sided communication protocols. Multiple timesteps 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 [1] are also being developed [78]. For visualization, a ParaView reader has been created to take advantage of the different ADIOS methods. Steering simulations with ADIOS does not seem to be supported as of today. However this I/O library does not present any theoretical limitations in its design for a possible extension.

A similar approach is provided with GLEAN [74]. The framework uses a client/server architecture to re-route data from a simulation to staging nodes. In this approach, the simulation is the client and the staging part receiving data is the server; 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.

The main advantage in these approaches is to have a staging area or coupled application located on concurrent resources, adding a minimal or null overhead on the simulation side (but effectively requiring an additional amount of resources to host the staging server or post-processing application, which is also the main issue of this approach). However, while in a tightly coupled approach, simulation has to wait for post-processing to finish before being able to compute the next time step, in a loosely coupled approach, since data is duplicated and staged into a remote memory, data analysis and visualization can be processed asynchronously. This is also interesting as heterogeneous architectures can be used together, one dedicated to the simulation and one to the post-processing, assuming that a relatively good network links both machines or node partitions together and that a common communication protocol can be used. Additionally, when a fault in

the post-processing application is detected, it can be guaranteed that the simulation will carry on computation and not be stopped.

Steering Considerations

As simulation and post-processing operate on their own copy of the data and in an asynchronous manner, several possibilities can be offered in terms of steering. While we have enumerated some of the possibilities that are offered by frameworks such as EPSN, other existing frameworks rely on a loosely coupled setting; they have been largely studied in [53] already.

One of the first computational steering environments that has been developed was the visualization and application steering environment (VASE) [14]. The VASE framework consists of a collection of programming tools and system software. Designed to work with existing codes written in Fortran and based on simple annotations in the source code, VASE tools construct a high-level model from the application, enabling the user to work with this model rather than at the detailed source-code level. Monitoring and steering are performed through break-point scripts. Scripts can read and write variables in the application, control the flow of data between processes (e.g., to send data to a visualization process), and call subroutines defined in the application program.

SCIRun [60, 61], a problem-solving environment, is another well-known framework. It uses an object-oriented data flow approach to enable the user to control scientific simulations interactively (synchronously or asynchronously) by varying boundary conditions, model geometries, and computational parameters. SCIRun was designed for the development of new applications, although it is possible to incorporate existing applications into the system. An application in SCIRun resides in one or more modules, implemented in C++. Writing a new module involves writing a new C++ class. The user interface of SCIRun includes several predefined modules for data visualization and program monitoring (progress meters, thread display, memory usage statistics). For user input, modules can integrate a Tcl/Tk user interface with which the steerable items of that module are controlled. SCIRun2 [79] now relies on a more loosely coupled setting and supports distributed computing through distributed objects. SCIRun2 is based on SCIRun and on the Common Component Architecture (CCA) [8], which aims at defining a minimal and standard set of interfaces for interoperability between components. Parallel components are managed transparently over an $M \times N$ method invocation and data redistribution subsystem.

CUMULVS [40] is another library that, in addition to providing access to distributed data at runtime (runtime tracking), also supports fault-tolerance to failures by using check-pointing mechanisms. The steering capabilities of CUMULVS include model exploration and performance optimization. CUMULVS originally aims at interfacing PVM programs but interoperates well

with simulations that use MPI or other parallel environments (and CUMULVS has also been integrated with the Global Arrays PGAS model [54] for distributed shared-memory programming). The basic principle of CUMULVS is to have the user declare in the application how an array or field of variables has been decomposed over a collection of parallel processors and specify which parameters are allowed to be modified or steered during the computation. To allow steering, the user interface process creates a loosely synchronized connection with the application, which guarantees that all tasks in the application will apply the steering updates at the same time or point in the application. To prevent multiple viewers from steering the same parameter simultaneously, a viewer can lock a steerable parameter by obtaining the steering token of that parameter.

Similarly, RealityGrid [12], which is mainly used for grid computing, provides an interface for computational steering. One can connect dynamically to the simulation, monitor values of parameters and edit 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. To make use of the steering library, an application must satisfy certain requirements. In particular, the application must have a logical structure such that there exists a point within a control loop at which it is possible to carry out steering tasks such as pause, resume, detach and stop, but also get and set values of steerable parameters. The computational steering API defined in RealityGrid is quite exhaustive and provides additional functionality, such as the ability of checkpointing/rewinding from a registered control point. A connected steering client may then instruct the application to create a checkpoint and later restart from that particular checkpoint.

These frameworks all provide a comprehensive interface and several features such as steering of parameters, boundary conditions, model geometries. The loosely-coupled approach allows these operations to be performed synchronously or asynchronously without interfering with the simulation, allowing even fault-tolerance in the case of CUMULVS. However, it is worth noting that all share one common drawback, the data model used and user interface does not follow the original data model used to perform I/Os and in most cases, the simulation code will require significant rework (by defining steering modules, etc). Again, it is possible to go further and define a generic interface for both the simulation output and computational steering, allowing data to be exchanged over the same existing interface, thus minimizing the modifications in the source code and allowing compatibility with existing data models and file formats (which is critical for visualization purposes, interoperability, etc).

2.2.3. Hybrid Approach

In a hybrid approach, tightly coupled and loosely coupled techniques are intertwined so that one benefits from both approaches by first reducing data into a tightly coupled setting before sending it to a concurrent resource for final post-processing and visualization.

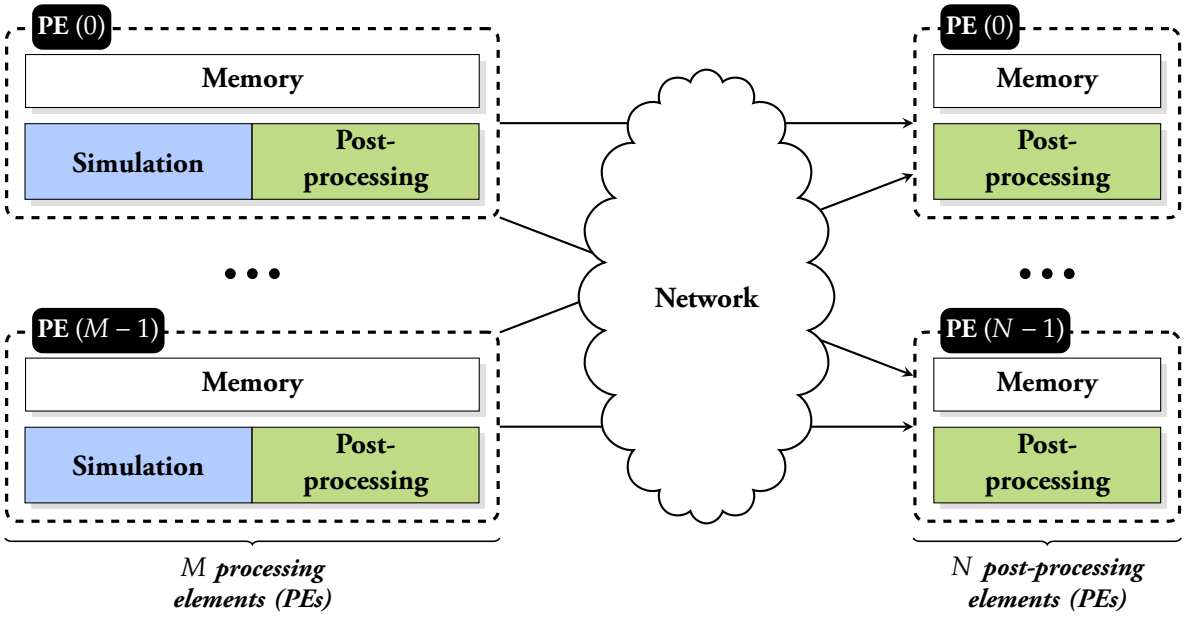


Figure 2.9. Both previous techniques are combined and only a tightly coupled setting is sent through the network.

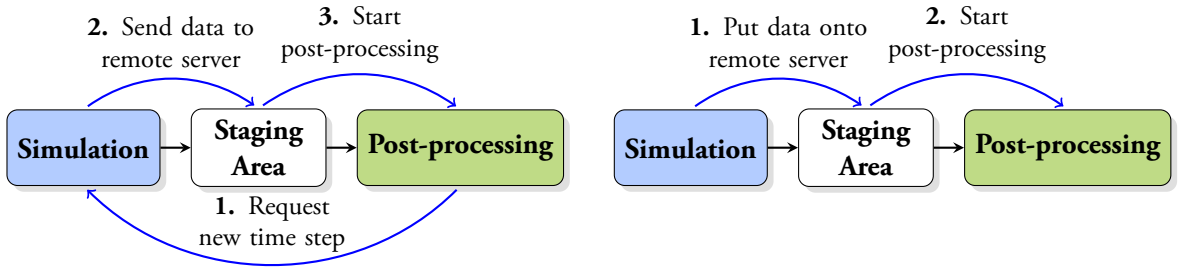
A good illustration of this approach is made by using both ParaView co-processing and GLEAN frameworks [28], removing the need to use disk I/Os to write (partly) post-processed data. By using a GLEAN writer in the in-situ post-processing pipeline, the GLEAN client (on the simulation side) re-routes data to staging nodes. The staging nodes host both a ParaView server and a GLEAN server. A VTK GLEAN reader is used to read data from the GLEAN server, and convert it into VTK objects, without copying memory. Additional ParaView filters can then be applied to convert data into the final post-processed data that is to be visualized.

To a lesser extent, this approach shares the drawbacks of both previous techniques, a non-negligible overhead is present on the simulation side (memory and code annotation) and data transfers depend on network capabilities and communication protocols. Additionally, depending on the framework used, it may be complex for a non-expert user to implement. This approach however gives a good compromise if transferring the whole raw data is not possible or if the data analysis needs to be done on the same nodes as the simulation.

2.3. Our Push-driven and Loosely Coupled Approach

The approach that we propose to define uses a loosely coupled model and can be defined as an in-transit visualization approach (as opposed to an in-position visualization approach since data is sent to concurrent resources). Even on recent systems, memory consumption is an important issue and transferring data to a remote system or node partition may be the only way (depending on the analysis algorithms that are to be performed) of not decreasing the solver performance. Therefore the simulation pipeline in our approach is split into three different components, the simulation that outputs data, the staging area that receives data from the simulation, and the post-processing application that analyzes data coming from the staging area.

2.3.1. Push-driven Transfers



(a) The simulation receives a post-processing request, a new time step is sent that updates the pipeline.

(b) A new time step sent from the simulation updates the post-processing pipeline automatically.

Figure 2.10. Two approaches for in-transit processing.

To transfer data between simulation and post-processing applications asynchronously, we distinguish two techniques, push-driven from pull-driven. In a *pull-driven* model, as shown in figure 2.10a, it is common to have the simulation defined as the server and the post-processing and staging application defined as the client. The post-processing application sends a request to the simulation, asking for a new time step. At the end of a computation step, when the simulation sees that a request is present, data results are sent to the staging area (or to the post-processing application). To be able to handle the different requests and transactions, an additional service (thread or process) needs to run on the server, as all the requests must be non-blocking for the simulation. To go further in the direction that we have chosen, and not interfere at all with the solver (i.e., not create any additional overheads), we choose to adopt a *push-driven* model where data is pushed and put into a remote memory. In this case (see figure 2.10b) the post-processing/staging application is the server and the simulation is the client. At the end of a time step, data is pushed and staged to

remote nodes and is then automatically post-processed. Note that if post-processing data is slower than computing a time step, simulation may be slowed down but in this case one can imagine to store multiple time steps in the staging area or potentially skip some of them. In the following sections, only the case for storing a single time step will be considered.

2.3.2. In-memory File Exchanges

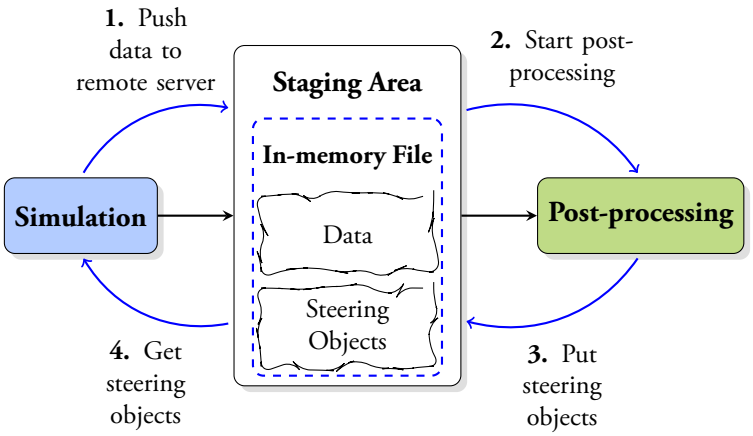


Figure 2.11. Transfers between post-processing and simulation occur through in-memory files where data but also steering data can be stored.

As we want to be able to interface with multiple codes that can potentially use different data models and different data representations, we decide to use in-memory files (that follow a hierarchical pattern) as a *generic* interface, to exchange and store data in the staging area. For steering the simulation, this approach will also allow us to exchange not only parameters but also blocks of data using a reserved section of the memory mapped file, which is illustrated in figure 2.11. Memory files are here represented in a very simple form but they have to be actually mapped onto a more complex memory system so that exchanges and transfers are performed as fast as possible between the different components (the main bottleneck of this approach must be minimized). To map the files and handle the transfers, different strategies and techniques will be presented in the next chapter.

2.3.3. Distributed Shared Memory for Data Staging

The staging area in our approach is a distributed shared memory (DSM) [17], which we also use as a steering interface as presented in the previous section, and is distributed among N processes. This is not a novel idea, other recent approaches such as the one from Lorenz et al. [46] make

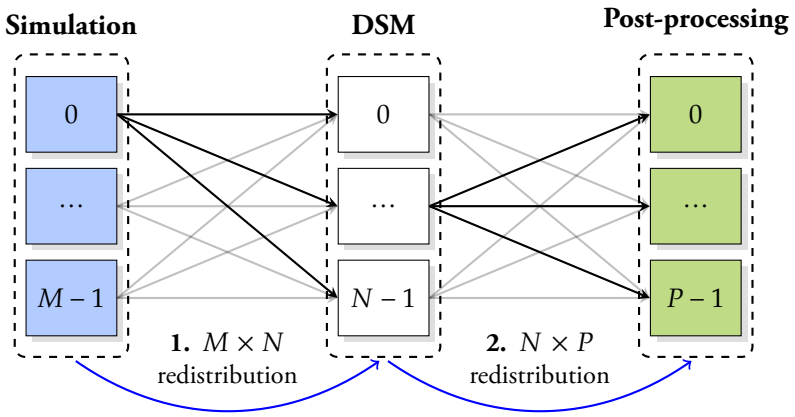


Figure 2.12. Parallel data transfers between the DSM interface and the post-processing application follow a $M \times N \times P$ redistribution.

use of the DSM concept and have formalized consistency models and protocols to ensure data integrity within a steering environment. However the novelty and originality of our model differs from theirs by the push-driven architecture and the hierarchical file approach used to map data and steering objects onto the DSM.

As described in figure 2.12, the solver component is distributed among M processes, the staging area among N processes and the post-processing among P processes, and it is common to have $M \gg N$ (for the reasons explained in 2.1.2). This results in a $M \times N \times P$ data mapping between the different components and we will present in the next chapter the different approaches used to map and transfer data between them. Note also that whereas the staging area and post-processing components can be located on different sets of nodes or machines, they may also be part of the same post-processing server application and in most cases, N may be equal to P.

2.3.4. Extension to I/O Libraries

Generally, from an implementation point of view, simulation users and developers spend a non-negligible effort in implementing interfaces for outputting data, making use of I/O libraries such as HDF5 or NetCDF, as described in 2.1.1. In a loosely coupled approach I/Os need to be re-routed to a staging area and as a consequence, the simulation I/O interface may be subject to several modifications.

In our approach, we want these modifications to be as minimal as possible and not have to rewrite the whole I/O interface of the simulation code. This objective can be reached by extending the I/O library itself, so that the user may be able to use the same API and I/O interface but, as

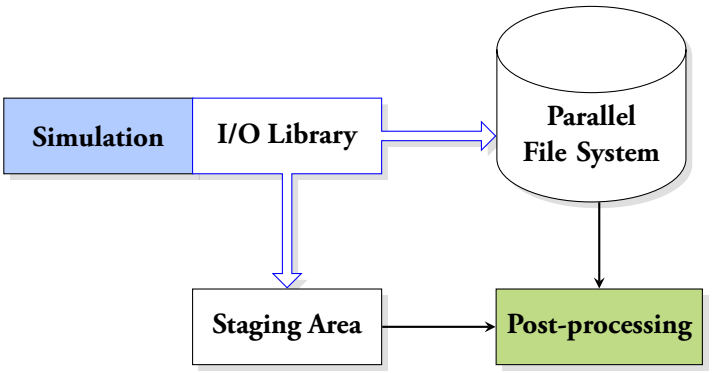


Figure 2.13. Using a unique I/O library, a given simulation code can switch from writing to a parallel file system to another I/O mode, which is to re-route data to a staging area.

illustrated in figure 2.13, will see its data being re-routed to the staging area. This also presents two advantages: keeping the ability to archive data, so that one can potentially combine both approaches, traditional and in-situ processing, and keeping the hierarchical approach provided by I/O libraries so that data can be easily accessed. We will present in section 4.1 how we make use in our implementation of the existing and widely adopted HDF5 library to achieve this goal.

2.3.5. Methodology

Based on the previous statements, we present in chapter 3 the detailed architecture of this approach and its inner mechanisms: in section 3.1 the communication system along with the synchronization and notification mechanism, in section 3.2 the in-memory file space layout and redistribution techniques used to transfer data to the DSM, in section 3.3 the exchange interface and required steering properties (i.e., the mechanisms used to exchange steering commands and data) and in section 3.4 how we deploy the different components of our architecture.

In chapter 4, we discuss the implementation choices that we make to achieve our architectural objectives: in section 4.1 we describe how we make use of the HDF5 library to re-route data in parallel and show, based on the communication and redistribution strategies, the performance achieved on different types of systems. In section 4.2, we describe the post-processing part of the implementation, necessary to read and interact from/with the DSM, as well as the associated performance.

Validation test cases are then studied in chapter 5. We present in 5.1 and in 5.2 the integration of both in-situ visualization and steering features of our framework into two simulation codes from the NextMuSE European project [24]. We demonstrate how our DSM approach extends the capabilities of these codes in order to solve genuine engineering problems.

Chapter 3.

A Loosely Coupled Model: Architecture and Requirements

As described in 2.3, the approach that we chose follows a loosely coupled model. In this chapter, we discuss the different architectural choices that we make to meet our two main requirements: one, avoid overhead on the simulation side; and two, provide a two-way communication model for the exchange of data. Whereas one would only need one-way exchanges for simple in-situ visualization, for simulation steering a two-way communication system allows the reception of commands from the simulation and therefore the definition of a comprehensive steering interface.

Architecture Overview

As illustrated in figure 3.1, the architecture we wish to define is organized as follows: a simulation is coupled to a post-processing application through a distributed shared memory (DSM); the data itself is stored in an in-memory file that is distributed over the network. To be able to generically use and read data stored in the file (see 2.1.1), information about the internal data hierarchy and layout must be cached: this piece of information is called *metadata*. An in-memory file must be consequently composed of two separate parts, one containing the *data* itself and one containing the *metadata*. Additionally, steering commands and steering data can be stored into a separate chunk of the file.

Simulations can generally be decomposed into four main steps: an initialization step, a solving step, an I/O step and a finalization step. The I/O step may write results of the previous solving step before looping back to the next computation step depending on the number of steps requested (which may be time steps if the simulation loops over time). In a typical processing loop, instead of writing to disk during the I/O step, the simulation may reroute data in parallel through the network to the DSM. This is the critical point of the architecture: data must be sent using very

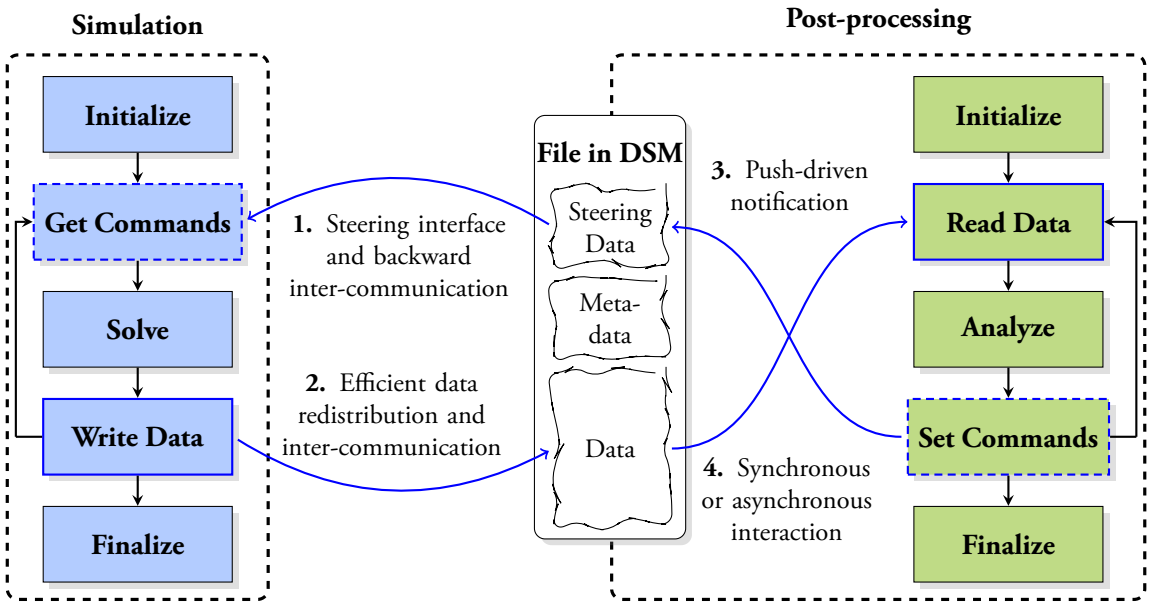


Figure 3.1. Data from the simulation is re-routed to an in-memory file stored in DSM, analysis starts in a push-driven fashion, steering data is sent back through DSM and read back from the simulation.

efficient communication techniques so that the network bandwidth is maximized. This will be more detailed in section 3.1. Additionally to be more efficient, data stored in the DSM is sent and stored using different mapping and redistribution patterns. This is detailed in section 3.2.

When the simulation writes new data into the DSM, we want the post-processing application to automatically start analysis without having to do any DSM status polling. This means that a fully push-driven structure must be used instead, along with a complete notification system to forward possible events to the post-processing application; more details are presented in section 3.1.5. When data from the simulation is sent to the DSM (synchronously or asynchronously depending on the communication system used), the simulation can carry on computing without waiting for the analysis or visualization steps to be completed, as another copy of data is remotely created and stored in the DSM.

Once the data is analyzed, a user may—or may not—give new steering orders to the simulation. To achieve this goal, we present a comprehensive steering interface in 3.3 allowing simple commands and more complex data to be sent back to the simulation. Steering orders are generally received by the simulation at the beginning of a new time step but we will see in section 3.3.4 that we can also define two different modes of interaction depending on user demands, one synchronous and the other asynchronous.

Finally, it is important to note that depending on the system and configuration that are to be used for running the simulation and the post-processing applications, different options for deploying the components of the architecture may be chosen. This is detailed in 3.4. To optimize the use of resources on the systems and not do unnecessary memory copies, as illustrated in figure 3.1, the DSM may be *co-located* with the post-processing application, or on *separate* computation nodes if the data cannot be stored on *same* nodes as the post-processing application.

3.1. Communication Interface

The communication interface defined in this section is one of the crucial points of the architecture as all the exchanges and operations between the simulation and the post-processing application must be executed as fast as possible. The simulation writes data packets in parallel to the DSM, which become distributed not only spatially but also temporally as the different pieces may transit at various speeds through the network. Therefore the operation order between the simulation and the post-processing application must be preserved to ensure not only event ordering but also data integrity.

3.1.1. Communicators

The DSM, which is the staging area that we previously defined in 2.3, can be seen as an addressable remote memory space from the simulation point of view. To transfer data in an efficient manner from a given parallel simulation code that uses M processing elements to a DSM that uses N processing elements, we need $M \times N$ links so that the local simulation data can be sent to any address of the remote memory. This point-to-point communication model may also be the only way of maximizing the network bandwidth depending on the number of physical links available between the simulation and the DSM. For communication within or between processes, we distinguish two types of communicators[1]:

1. An **intra-communicator** represents the communicator used for internal communications performed by a given application;
2. An **inter-communicator** links two different applications or two different sets of processes together.

[1] The concept of communicator is very close here to the notion of communicator objects defined in MPI, where the basic group object can be seen as the application.

In our approach, the simulation uses its own intra-communicator for all the internal data exchanges and an inter-communicator to communicate with the DSM. The DSM has also its own intra-communicator, which may be used for internal synchronization purposes or may be shared with the post-processing application if the DSM and the post-processing application share the same global communicator.

If we consider the simulation code and the DSM as two distinct SPMD (Single Program Multiple Data) codes, the inter-communicator linking these two codes together must be dynamically created at run-time and must use a connection procedure so that one code may try to reach the other one to create a new inter-communicator. However if we consider the two codes as a unique MPMD (Multiple Program Multiple Data) code, the inter-communicator is statically created and shared during an initialization phase.

Once the inter-communicator is created, the simulation may send the actual data that needs to be written and stored remotely, but also commands or events, which may be requested for different reasons (which we will detail in 3.3). As one of our objectives is to interfere as little as possible with the simulation code by using a push-driven approach, we define the simulation as the client and the DSM as the server (note that as explained in the previous overview, the DSM may be combined with the post-processing application). To be able to communicate to the server and send commands at any time, a thread—referred as *service thread*—must be listening on the inter-communicator. This thread may therefore receive a command, treat it and carry on listening for new commands until disconnection.

To send the actual data we distinguish two communication modes, two-sided and one-sided. In a two-sided approach, the sender and the receiver must participate to the communication operation. In a one-sided approach, (as the name suggests it) only one side needs to issue a *write* or *read* call to achieve the communication. This communication mode has been introduced more recently and is exploited in libraries such as MPI [2].

3.1.2. Two-sided Interface

In a two-sided approach, as the nature of the events and transactions received by the DSM is not known in advance, data transfers (using communication libraries such as MPI) can only be performed in two phases, which creates a synchronous operation. As shown in figure 3.2, in the first phase, the sender sends a *send data* command to the DSM with the size of data to be transferred. In the second phase, the sender sends the data and the DSM receives it. For communication from the DSM to the simulation, the simulation sends a *receive data* command and the DSM sends

the requested data back to the simulation. The same mechanism applies for the post-processing application that queries data from the DSM.

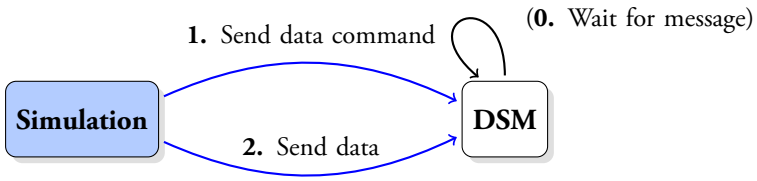


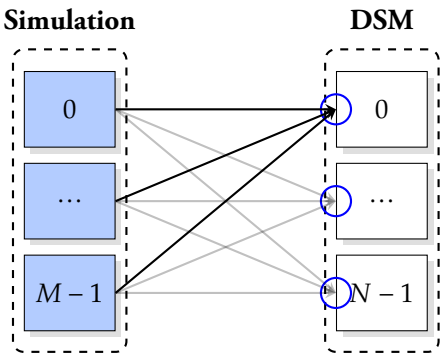
Figure 3.2. A service thread listens for incoming messages and data is transmitted in two phases.

This communication mode can be well-suited for a one-way communication approach (i.e., from the simulation to the DSM). However for a two-way communication approach where data needs also to be sent back to the simulation, it is necessary to be able to listen on the two communicators (inter-communicator and intra-communicator) at the same time, as requests may come from both ends. Therefore this requires *another thread* or an additional polling mechanism, which can forward the requests coming from the intra-communicator to the *service thread* (assuming the *service thread* is listening on the inter-communicator).

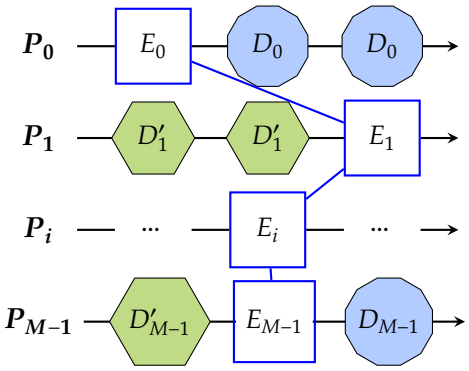
As we want to be able to preserve the order of the requests and not mix up commands and data, we need to define a synchronization mechanism that preserves the ordering of the commands, notifications and data, since one message sent in parallel from one link can possibly arrive before messages that were sent earlier on different links.

Synchronization

Compared to a traditional DSM architecture, the architecture that we define is required to handle not only a distributed shared memory for data staging but also commands and notifications used for our push-driven mechanism. Therefore in this two-sided approach, at anytime for each DSM process as shown in figure 3.3b, data and events may come from the M simulation links (see figure 3.3a). For instance, when the simulation has finished writing/sending data to the DSM, it may send a notification (see 3.1.5) to tell the DSM (and in turn the post-processing application) that new data has been produced. This notification event must be forwarded to all the DSM processes to become a global event, as every process must have knowledge of the global state of the DSM. Moreover, as data can arrive from the M simulation links to any DSM process, it is important to note that separating data from events does not change anything in this approach as the parallel message ordering would still not be guaranteed. Therefore an event sent to the DSM must be a global and synchronizing event.



(a) Each DSM process has M communication channels to receive data and/or events from the simulation.



(b) Data stream coming from the simulation to one process of the DSM; only the ordering of events on a single communication channel can be guaranteed.

Figure 3.3. Two-sided event and data synchronization mechanism.

Algorithm 3.1 Synchronization of DSM communication channels.

Require: $x \leftarrow E_a$ and $a \in \{0, \ldots, M-1\}$
Ensure: $R = \emptyset$ and DSM data integrity is preserved

1. $p \leftarrow a$
2. $R \leftarrow \{0, \ldots, M-1\}$ {set of processes from which events must be received}
3. **while** $R \neq \emptyset$ **do**
4. **if** $x = E_p$ **then**
5. $R \leftarrow R \setminus \{p\}$
6. $(x, p) \leftarrow$ receiveData(R) {receive new data from any process in R}
7. **else**
8. process(x)
9. $x \leftarrow$ receiveData(p) {continue to receive data from p}
10. **end if**
11. **end while**

We note E_i the received events (which can be notifications or commands), D_i and D'_i the distributed data pieces that belong to different data blocks D and D'. As illustrated in figure 3.3b, each channel may receive a different number of data chunks, for instance in blue from the simulation and in green from the post-processing application, and events may arrive at different times. Note that, to guarantee a global ordering between events and data, the same event is sent to the DSM at the same time on every channel so that data chunks that belong to different data sources are not mixed up between events.

This synchronization problem can be seen in several parallel and distributed algorithms and a very similar approach is for example presented to aggregate multiple associated events through a hierarchical communication structure in TBONs [36] (Tree Based Overlay Networks). Algorithm 3.1 allows us to synchronize the DSM communication channels and preserve event ordering. To aggregate the global event E that is to be received by every communication channel, we construct a set R that contains the different process identifiers and take out from this set the process (here a) that has received the event E_a. We then wait for new messages coming from other processes that belong to R until we receive data from a new process p. We receive and process data requests coming from this process p until the same event E_p is encountered. We repeat this operation until all the processes have been treated and the set R is reduced to $\emptyset$. The communication channels are then synchronized and the global event is received.

Limitations

This approach is obviously not the best approach one can find to stage data in a push-driven fashion since it requires: an extra thread or message polling mechanism to handle two-way communications; the sending for every data transaction of an additional data request command; the synchronization of the communication channels in a tedious (but simple) way between the events.

Adding a thread to maintain a dynamic two-way mechanism is not an important limitation in the sense that processors and operating systems can now support several concurrent threads running at the same time without any real performance penalty. However the two other limiting factors of this approach may create a performance drop, especially at scale. While running on a few processors, the additional synchronization and transactions may only have a little impact and create a small overhead, but scaling up to thousands of processors will proportionally increase this overhead, which may become a real bottleneck.

3.1.3. One-sided Interface

In a one-sided approach, data transfers can be directly issued from the simulation to the memory of the remote DSM without any prior exchange request (as opposed to a two-sided exchange protocol). However to be remotely accessible, depending on the implementation used, remote memory descriptors need to be gathered from the DSM so that the simulation has knowledge of where to put the data to (note that this step may sometimes be hidden by the API implementation). Once the sender has gathered this information, data can be transferred and put at the requested address in a one-sided manner. As opposed to the previous approach, transfers can even be asynchronous, allowing the simulation to carry on computation without needing to wait for reception of messages, as no handshaking is required. The additional thread used in the two-sided approach for handling two-way accesses to the DSM is here no longer required as the remote end (i.e., the DSM) does not need to be contacted before sending (except for memory descriptor exchanges as explained above).

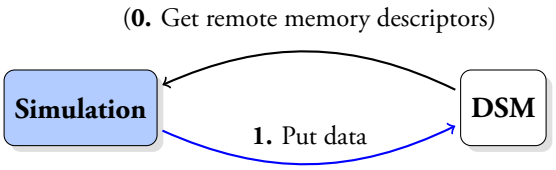


Figure 3.4. One-sided data transfer mechanism, remote memory descriptors may be gathered before put operations can be executed.

Also it is important to note that one-phase transfers presuppose that the remote DSM allocation remains the same during a simulation run, as re-allocating the memory would otherwise result in multiple memory descriptor exchanges, which may be an expensive operation depending on the underlying implementation. As a consequence, this one-sided communication protocol is more likely to be used for the exchange of data (*single* large memory allocation for the whole simulation run) and not for the transfer of notifications and events (*multiple* small memory allocations). Moreover transfer of events would require active participation of the remote end, as a received event must trigger an action specific to this event (see 3.1.5), which would imply memory polling or active synchronization mechanisms to be used. Therefore making use of a one-sided approach for the transfer of notifications and events does not bring any strong advantage to our model and is not considered in the following sections.

Synchronization

The remotely allocated memory needs to remain in a coherent state and remote *put* and *get* operations to the file previously created and distributed among the different DSM nodes cannot happen without any synchronization or locking mechanism. Part of this synchronization and locking mechanism can be implemented in the one-sided communication protocol itself, but the other part must be taken care of by the DSM architecture.

Before starting an access epoch, the defined remote memory *window*[2] is locked so that other processes that do not participate to the transfer cannot access it at the same time. Every operation that involves file modification or that is subsequent to a file modification will require a collective window synchronization, so that the file metadata and the data (being modified or read) are valid. Hence, when the in-memory file is opened or created, a global synchronization on the memory window is performed if the previous operation has modified the file metadata. With this synchronization mechanism, transfers only need to complete when the memory lock is released, and this creates a potential *asynchronicity* of operations (depending on the implementation used), which gives the simulation the ability to carry on computation and not waste time in transfers.

Whereas it is necessary, using *send* and *receive* operations, to guarantee that every process has finished sending or receiving data before the effective close of the file and the beginning of another operation; being able to synchronize on the entire window gives us here a much more flexible solution than a two-sided approach.

Limitations

Although this approach seems to be a more scalable approach compared to a two-sided approach, it requires the remote memory to be already allocated. This can be a potential implementation issue as it imposes a static memory model and not a dynamic one. If the post-processing application and the DSM run on separate resources, the DSM will be likely allocated as a static object. However if the DSM and the post-processing application share the same resources, it may be useful to have a dynamic allocation model where memory objects are automatically allocated and shared with the post-processing application so that memory descriptors and addresses can be shared between both applications. We will not consider this model in the following section (mainly because of the constraints imposed by the one-sided model) and instead consider a static DSM model.

Moreover, an inner limitation of this approach exists in its implementation, especially with the current MPI 2 one-sided interface that does not provide a very strong flexibility (mainly to prevent

[2] We use here the term of window [2] as only memory subsets may be accessed, even if most of the time this subset may be equal to the size of the entire memory that is allocated in the DSM.

users from non-coherent accesses) and we will see in more details in section 4.1.3 what the current issues are.

3.1.4. In-Memory File Access

For *in-situ* visualization and analysis, it is assumed that the simulation will write to the DSM and the post-processing application only read from it. For steering, the simulation may write to the DSM and at the same time, the post-processing application may try to send user data. When post-processing and DSM are decoupled and located on separate resources, the restriction imposed by the communication and synchronization system is sufficient to prevent the DSM from multiple and concurrent memory accesses. However, if post-processing and DSM share the same resources (which is the most common use case), multiple memory accesses to the same resource imply addition of a *locking* mechanism, completing the communication and synchronization system previously introduced.

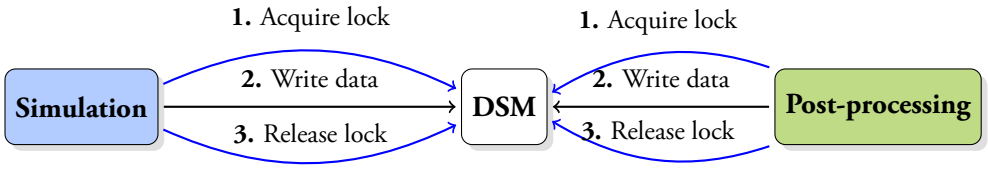


Figure 3.5. Additional file locking mechanism, any access locks the file to prevent the shared memory from concurrent memory accesses.

As described in figure 3.5, 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 will become available to the coupled process.

3.1.5. Event Notifications

In a common scenario, the simulation makes periodic writes to the DSM and can, when using the steering API, make reads to see if any new data or instructions are available. The notification mechanism is illustrated in figure 3.6 describing the most common use case where the DSM and the post-processing server share the same resources.

In a pull-driven system, the analysis code must query whether new data is present and if so update its analysis pipelines. Instead in a push or event-driven approach, notifying the post-processing application when new data is produced works in several stages as follows. Each DSM server process has a constantly listening *service thread* (see 3.1.1) that receives and treats internal

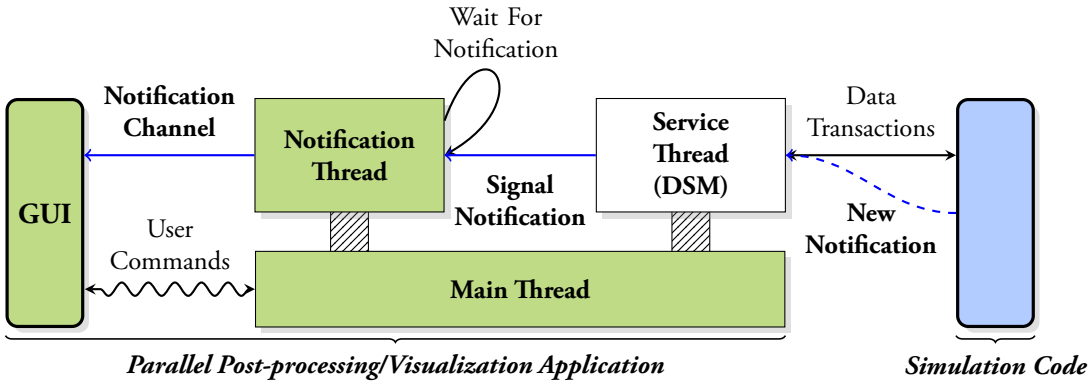


Figure 3.6. Thread and push-driven notification mechanism used to inform the GUI and the DSM of new events. When a new data or new information notification is received, the task relative to this event is performed and/or the associated post-processing pipeline is updated.

data transactions. When the file is closed by the simulation, a notification is sent to the DSM server and is picked up immediately by the *service thread*. This notification may then be forwarded to the main post-processing application process (if the post-processing application is parallel) or/and to the GUI process to trigger specific actions (see 4.2.1). To achieve this, depending on the post-processing application architecture, another thread, referred as *notification thread*, may be woken (only on rank 0 of the parallel post-processing application or application hosting the DSM servers, as events are globally visible) to then send a notification event to the GUI. When the GUI is notified, the task corresponding to the received notification code is then performed in the DSM user interface.

Note that in some cases (examples are detailed in 4.2.1), it may be useful for a user to send notifications/events from the simulation code to the DSM (and in turn to the post-processing application) not only when the file is being closed but also when the simulation reaches specific points, so that corresponding actions can be performed in the GUI. While the post-processing application may be busy performing other tasks, this two step process from *notification thread* to GUI and back to post-processing *main thread* ensures that the we do not trigger the execution of post-processing tasks during another user-driven event on the same post-processing *main thread* inside the analysis tasks (which would in this case create a bottleneck in the post-processing workflow). It is important to note also that the notification mechanism is one-way only. When the post-processing application writes data to the DSM, no signal should be triggered in the simulation as there is no *service thread* running on the simulation side (we do not want to create any overhead of any kind on the simulation side).

3.2. DSM Mapped Files

Data and steering objects are stored in a file that is mapped onto a distributed shared memory. While the previous communication system and the notification and synchronization mechanisms allow us to exchange data and steering objects, we show in this section how one can dynamically redistribute data that is sent to the DSM so that the network bandwidth is maximized.

3.2.1. File Memory Space Considerations

In our architecture the DSM is distributed among N processes, each process allocating l bytes of data, which gives a total DSM length of $L = l \times N$. Using a linear addressing, the DSM is contiguously filled from process rank 0 to process rank $(N - 1)$.

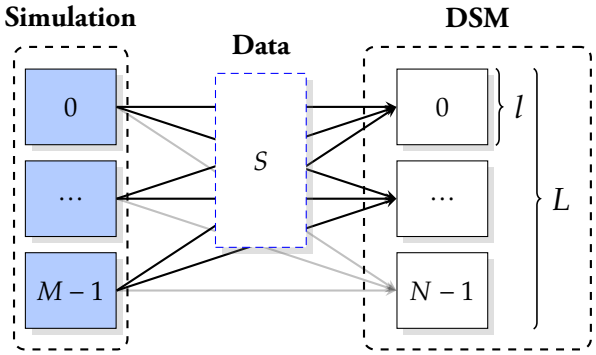


Figure 3.7. The DSM is distributed among N processes and has a total length of $L = l \times N$, each process allocating l bytes of data. Using a linear addressing, only $\left\lceil \frac{S}{l} \right\rceil$ processes are used to receive data chunks of size S.

As shown in figure 3.7, if a simulation writes a file of size S, the actual number of processes used to receive data will thus be $\left\lceil \frac{S}{l} \right\rceil$ with $S \leq L$. This method can provide relatively good performance when $S \simeq L$; if the file written is composed of several different datasets (each much smaller than L), which are contiguously (and sequentially) mapped onto the DSM, individual simulation processes will waste bandwidth by using only a small partition of the network links available. Then the simulation may write either a single large data chunk, which will be sent in parallel to the DSM using all the links available, or multiple data chunks, each one at a time using a partition of the links available. We therefore sought better strategies that can be enabled on demand.

3.2.2. Redistribution Methods

In this section, we focus on three different redistribution strategies, from the most simple to the most complex strategy: mask redistribution, block-cyclic and random block. These strategies are general strategies and only affect the redistribution of data packets, and thus only modify the DSM address mapping. Note that more advanced and specific strategies that would apply to the dataset objects themselves could be defined as well but this is not the purpose of this section.

Mask Redistribution

When $S \ll L$, a first simple strategy is to automatically re-size the DSM window to the requested file size without any concrete memory free or reallocation. As described in figure 3.8, if S is the size of the data that is to be written, N the number of DSM processes and l the local buffer size, a mask of size $l - \left\lceil \frac{S}{N} \right\rceil$ will be applied to each buffer, which reduces the overall DSM size to $\simeq S$. Since the data that is sent fits into the memory perfectly, all the DSM links between the simulation and the DSM will be used for receiving.

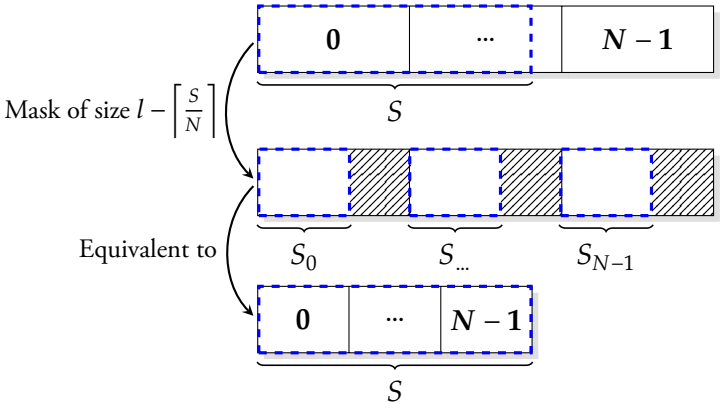


Figure 3.8. Using a mask redistribution, the DSM memory space of size L is virtually reduced to $\simeq S$.

However, as shown in figure 3.8, even if applying a mask to every local DSM buffer can effectively improve the overall bandwidth by making the local buffer size l equal to $\left\lceil \frac{S}{N} \right\rceil$, which in turns makes $L \simeq S$; it does bring two main drawbacks: the most evident one is that it wastes memory allocated on the DSM, the second one is that it does not solve the multiple dataset problem mentioned above as the newly created memory space can be seen as a contiguous memory space where data is linearly mapped. Therefore, writing multiple datasets of size S (where $S \ll L$) means again writing to only a small portion of the DSM processes available at the same time. Hence this solution can only be optimal when only one large dataset is being written.

Block-Cyclic Redistribution

The second strategy to be considered is a block-cyclic redistribution [75]. It is a simple strategy and it potentially allows a good data load balance between DSM processes. A block size s being fixed, the DSM address mapping is decomposed into $\frac{L}{s}$ blocks. For convenience, the DSM length L is adapted so that it becomes a multiple of s. Blocks are distributed in a round-robin fashion: the B^{th} block is assigned to the process rank ($B \bmod N$). Hence every address a is associated to the following triplet (n, o, i), which can be written as

$$a \mapsto \left(B \bmod N, \left\lfloor \frac{B}{N} \right\rfloor, a \bmod s\right)$$

the first term n being the process index within the DSM, o the local block offset in a process and i the local address offset within a block.

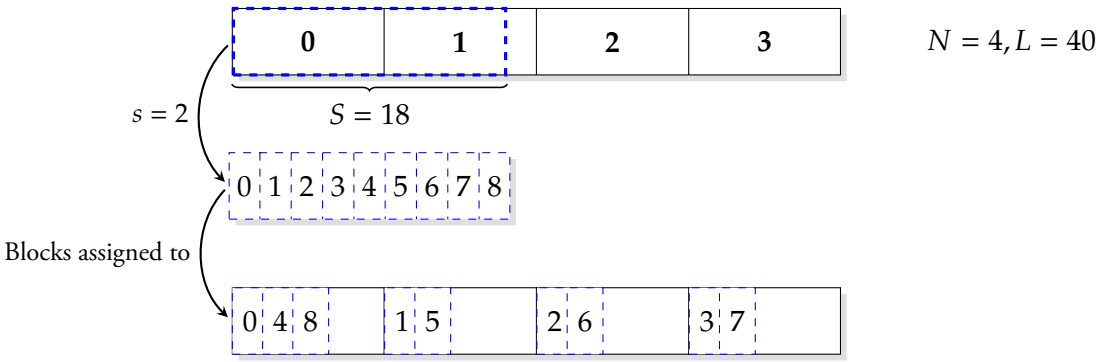


Figure 3.9. With a block-cyclic redistribution, data distribution is load balanced over the entire DSM using all the network links available.

As illustrated in figure 3.9, a block of size S written to the DSM (according to a block-cyclic addressing) is decomposed into smaller data chunks of size $\frac{S}{s}$. In this example, with $S = 18$ and $s = 2$, 9 data chunks are created. These chunks are then redistributed in a cyclic manner between the DSM processes. With 4 DSM processes, block 0 gets mapped to process 0, block 1 to process 1, ..., block 4 to process 0, etc. This method presents two main advantages: bandwidth is not wasted even if $S \ll L$; data chunks are well load balanced, which is especially beneficial when multiple datasets are written. However this method can potentially create a huge number of data transactions, which can result in a performance drop (as we will see in section 4.1.4) depending on the network communication protocol and block size that are used.

Random Block Redistribution

The third strategy considered consists of re-using the algorithm previously described, scattering the DSM address space into pieces of size s. Another step is then added to the redistribution pipeline, shuffling the blocks using a mapping vector (so that blocks can be retrieved). As described in figure 3.10, in this example, 9 blocks of size 2 are created from a data chunk of size 18 (which would be written using only a half of the DSM links available with a standard contiguous redistribution). In this strategy, instead of redistributing blocks in a cyclic manner (as opposed to the previous solution), blocks are shuffled before they get sent to the DSM processes, which gives the following mapping: block 0 gets mapped to process 0, but then block 1 gets mapped to process 0 as well and then block 2 gets mapped to process 1, etc.

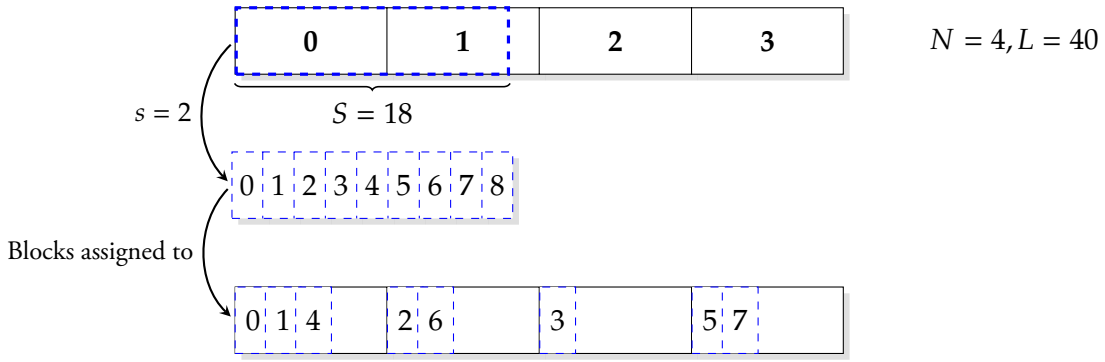


Figure 3.10. With a random block redistribution, data is randomly balanced over the entire DSM, potentially using all the network links available and potentially avoiding distinct simulation processes to write to the same DSM processes at the same time.

This method can present some advantages compared to the previous solution (but keeps the same main drawback): it may avoid a possible network congestion if two simulation processes were sending data to the same DSM process using the block cyclic redistribution algorithm. This may occur with a periodic frequency introduced by certain communication patterns and data distributions in the file, especially when the DSM makes use of a small number of processes compared to the number of simulation processes.

With these two last strategies we are effectively able to well load balance data over the DSM processes, even when multiple datasets are being written. However, as we will see in section 4.1.4, the performance of these methods is highly dependent on the architecture and on the underlying network protocol that is used.

3.3. Exchange Interface

Based on the previous mechanisms, data can be mapped and sent to the DSM in an efficient and optimized manner. In 2.3.4, we proposed to model the DSM I/Os as an extension to existing I/O libraries. In this section, we show how we take advantage of this property and use a hierarchical approach to store data into the DSM. While in-situ visualization is straightforward as data is automatically re-routed to the DSM, application steering is not possible without some fundamental changes of the simulation code to respond appropriately to changes being sent in. We present in this section which requirements need to be defined to send a comprehensive set of steering orders back and forth between the simulation and the post-processing application. We also present in 3.3.4 two different operating modes, one synchronous and the other asynchronous.

3.3.1. I/O Interface and Hierarchical Data Model

To write and read data, simulation and post-processing applications make heavy use of existing I/O libraries such as HDF5, netCDF, which are presented in 2.1.1. Using a hierarchical data model such as the one that exists in these I/O libraries, data can be organized so that complex meshes or data along with their associated values can be easily written and retrieved. Therefore a given data structure or *grid* can be represented using *groups* along with *datasets*, multidimensional arrays of data elements, which may for example represent connectivity and coordinates of a mesh, etc. Typically data stored in a *file* can be represented using the following pattern:

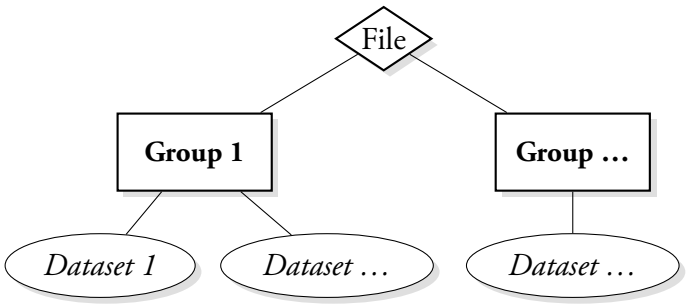


Figure 3.11. Hierarchical representation for storing data.

A file can be composed of multiple groups, which can contain themselves multiple datasets. Using the existing hierarchical data model of I/O libraries, one can organize data into groups and datasets. Applying this to our DSM interface, a simulation code can perform reads and writes using the existing I/O API (but all the traffic is re-routed to the DSM as we described in 2.3.4), which basically consists of the following calls: *create file*, *create group*, *write data*, *read data*, etc. Assuming that a description of the hierarchical representation is provided (see 4.2.2), the post-processing

application, when told that new data has been written, will use the same I/O interface to read data back from the DSM. This low-level interface allows simple exchange of any type of data structure. For simulation steering, specific control on the type of data that is to be exchanged and on its location needs to be given. In this case, a higher level interface will be used as we will see in 3.3.3.

3.3.2. Required Steering Properties

One of the first requirements when steering an application is the ability to change a simple scalar parameter. As previously described, since our interface is built on top of existing I/O interfaces that follow a hierarchical data model, it is trivial to store such a parameter as an *attribute* or as a singleton in a *dataset* within the file. However, to be easily retrieved by the simulation, we want this parameter to be stored in a special section of the file, which we call *Interaction Group*. Adding support for vectors requires only the use of a *dataset* in the file. Once the ability to write back a *dataset* exists, it is easy to extend support to handle point arrays, scalar/vector arrays and all other types that are used within post-processing applications to represent objects. We are thus able to read from the simulation any structure that is stored in the file. Moreover, because data written by the simulation is stored in a file that follows a hierarchical data model, objects or existing datasets can be easily and independently modified from the post-processing application.

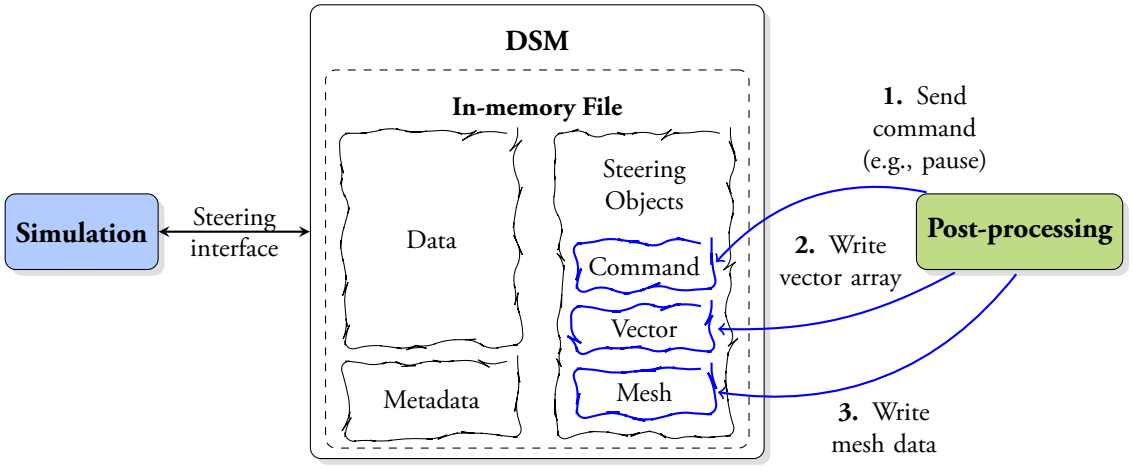


Figure 3.12. We need to be able to send back commands to the simulation (a pause command for example), simple arrays such as scalars and vectors to control simple parameters, but also entire meshes or complex data arrays to allow complete re-meshing of simulation objects.

As illustrated in figure 3.12, the post-processing application can transmit commands and data back to the DSM using either implementation specific calls (for all the commands that are specific to the steering implementation such as pause/play) or using the lower level I/O interface that performs reads and writes directly to the file. The simulation can then pick up these commands

or read the modified/new data using a steering interface that we define below. To be able to pick up a command, a parameter or a dataset, one crucial factor is that both sides of the transaction must be able to refer to the same shared/steerable parameter or dataset 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 4.2.3.

3.3.3. Steering Interface

In the context of in-situ visualization, while the simulation and post-processing applications can use existing I/O interfaces to access the DSM, we define for steering a higher level interface that can provide users with a comprehensive set of operations. As shown in figure 3.13, the interface directly interacts with the DSM interface and the I/O interface, wrapping lower level operations into higher level operations.

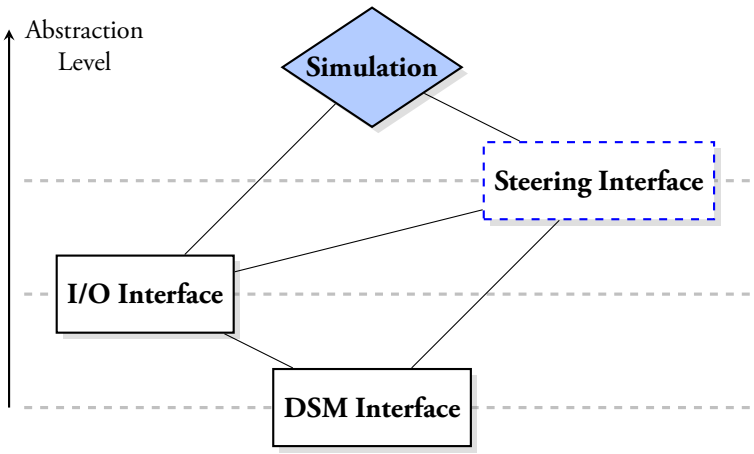


Figure 3.13. To access the DSM, a simulation code can make use of the I/O interface for direct read/write operations or can make use of the steering interface for more complex steering operations.

While most of the accesses are abstracted by the steering interface, it is possible in most cases to perform steering operations and send back data by using the lower I/O and DSM interfaces, but would require knowledge by the user of several internal DSM mechanisms (to send notifications etc). As one of our objectives is to provide users with a comprehensive interface that can hide the complexity of the underlying architecture and implementation, a more general and abstracted interface is defined for specific steering operations that can be directly used by the simulation code.

Interface Definition

To ensure most possible combinations and steering capabilities, we define the following operations:

```
(1)  init()
(2)  update()
(3)  wait()
(4)  scalar_get/set()
(5)  vector_get/set()
(6)  is_set()
(7)  begin_query()
(8)  end_query()
(9)  get_handle()
(10) free_handle()
```

As previously mentioned, all new parameters and arrays sent back for steering are stored by default at a given time step in an *Interaction* section, which is a *group* (see 3.3.1) created in the file that will contain all the interaction objects (commands, data, etc). In contrast with a visualization only use of the interface, when steering the simulation needs to be able to *read* from the file at any time (including at start-up for initialization data) and we therefore provide a steering library initialization call (1), which can be used to establish a connection between simulation and DSM before it may otherwise take place (i.e., at file creation, when accessing the I/O interface).

Once the environment is initialized, (4) and (5) allow the writing of scalar and vector parameters respectively, while (6) checks their presence in the file. The get functions are primarily used for getting arrays that have been passed from the GUI to the simulation, but the set functions may also be used by the simulation to send additional information to the GUI (such as time value updates at each step). Note again that raw calls to the underlying I/O library could also be used to read and write data but these higher level calls provide a more convenient and elegant way of accessing the steering data.

(7) and (8) are used when several consecutive operations are necessary, keeping the file opened between accesses. When accessed from the simulation side, file open and data requests result in *inter-communicator* traffic, which can be minimized. Particularly when the file is open in read only mode, metadata may be cached already by the underlying I/O library. In other words, by minimizing the number of file access requests, traffic may be correspondingly reduced.

(9) and (10) allow direct access to the I/O library *dataset* handle to the requested object (as the file follows a hierarchical approach) and this handle may be used with the conventional library API to perform I/O. The advantage of this is that a user can perform steering operations using the I/O library directly. For instance in the implementation described in 4 the full range of parallel I/O

operations and existing I/O features may be manually used by a user to improve the performance of read operations, which is particularly important if a very large array is modified (in parallel) by a post-processing application and returned to the simulation (e.g., object re-meshing).

Additional Synchronization

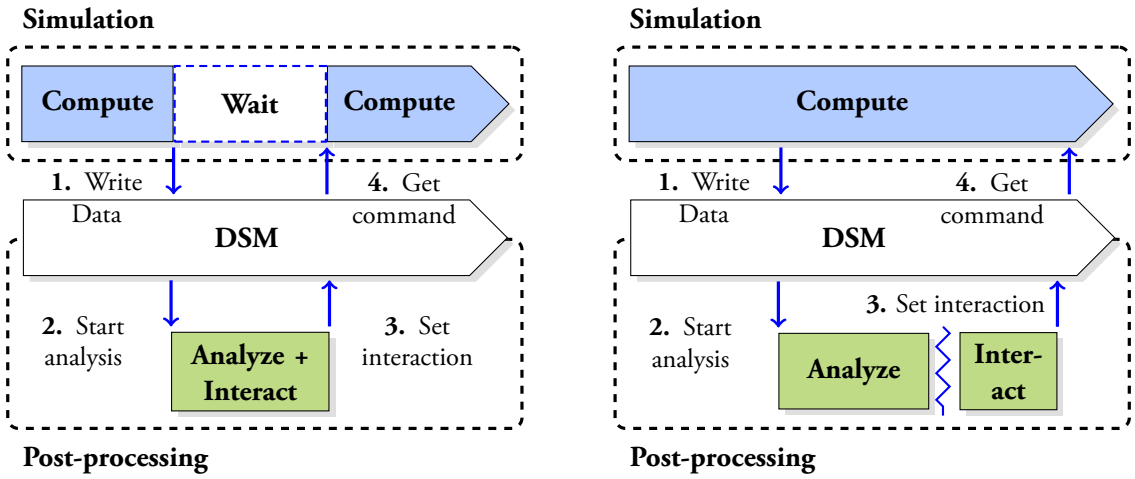
We also need to provide two different synchronization mechanisms, one using (2) and the other using (3). (2) allows the user to get and synchronize steering commands with the host GUI at any point of the simulation. It is a notification that can trigger updates and receive commands from the GUI and making use of this command actually sets synchronization points in the simulation code. For instance, when a pause request is set in the GUI, the code will stop at one of these points. Therefore, this synchronization is seen as *user dependent*.

(3) can also be used to coordinate the work-flow, making the simulation pause until some steering instructions are received. Additionally it is possible to forcefully *pause* and *resume* the controlled simulation, by locking and unlocking the file from the post-processing application side, thereby blocking the application at the next attempt to access it. The use of (3) can be preferred as it offers the chance to add wait and resume matching pairs of calls to the codes at arbitrary positions where the simulation should automatically pause to pick up new instructions. Therefore, this synchronization is seen as *implementation dependent*.

3.3.4. Timing of Interactions and Operating Modes

Using the synchronization mechanism previously introduced, the simulation may wait either for its data to be post-processed or for new commands and data to be returned. It may also check for commands (without waiting) to see if anything (such as steering commands and data) has been set for it to act upon while it was calculating. The two operation modes, referred to as *wait* mode and *free* mode are illustrated in figures 3.14a and 3.14b.

In *wait* mode, as shown in figure 3.14a, the simulation writes data (1) after each iteration and an analysis task is automatically triggered (2). Here the simulation waits for the analysis task to complete using a `wait` command so that the user can set new instructions and data (3). Finally the simulation re-opens the file and collects the commands set by the user (4). 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 and complexity of the analysis pipelines set up by the user to analyze the data read from the DSM and send commands or data back to it. Note that although the diagram in figure 3.14a shows no user interaction



(a) Wait mode. **(b)** Free mode.

Figure 3.14. Two principal modes of operation for timing of steering interactions.

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 to the DSM with different datasets prior to triggering a notification that makes the analysis start and is not limited to a single access as hinted by the diagram.

In *free* mode, as shown in figure 3.14b, the simulation computes without waiting and regularly writes data to the DSM (1), which triggers analysis tasks to be performed by the post-processing application (2). The user interacts via GUI controls (3) and new data is picked up whenever the simulation checks for it (4). The analysis is here overlapped with the simulation, which does not prevent it from accessing the file; the file locking and synchronization mechanism described in 3.1 ensures that either side can access it. The simulation is therefore delayed only by the time taken to check for new commands and data. In the absence of any new instruction, this time is of the order of milliseconds (see 4.2.3). Usually data will be read and the file will be unlocked immediately after, so that analysis will take place asynchronously. Based on this *free* mode of operation, the calculation can loop indefinitely issuing write commands, checking for new data using read commands. It is permitted to open, close the file in the DSM at any time (unless locked by the post-processing side) the simulation reaches a convenient point in its algorithm where new data can be transmitted. The post-processing side meanwhile, may receive a new data notification and immediately open the file to read data and perform its own calculations. Creating the interaction between post-processing and simulation is now entirely under the designer's control. A simulation that is operating in *free* mode must be capable of receiving new commands and data, and know that this data may not be directly related to the current calculation. As shown in figure 3.15, when a simulation

writes data of time step T into the DSM, the post-processing application starts analysis of this time step while the simulation has begun computing step $T + 1$ (assuming that we are talking about a simulation that iterates over time). During the post-processing phase, the user may set up different interactions, which are written into the DSM. There is no guaranty (in *free* mode) for those interactions to be picked up at time step $T + 1$ as commands may for instance be checked by the simulation only at the beginning of a new time step. At this point, the ability to send specific commands to the simulation that have special *meanings* or that contain time information becomes important (this is discussed further in chapter 5). Moreover as data sent back to the simulation is written into the same DSM file (and we only consider in this approach a DSM composed of one single file, see chapter 4), the user must take care in this mode of not erasing the file content when a new time step from the simulation is written. Note that although this problem could be solved if interactions were written into a separate DSM file adding *meanings* or time information to the commands would still be necessary to be sure that these commands are picked up when they need to be.

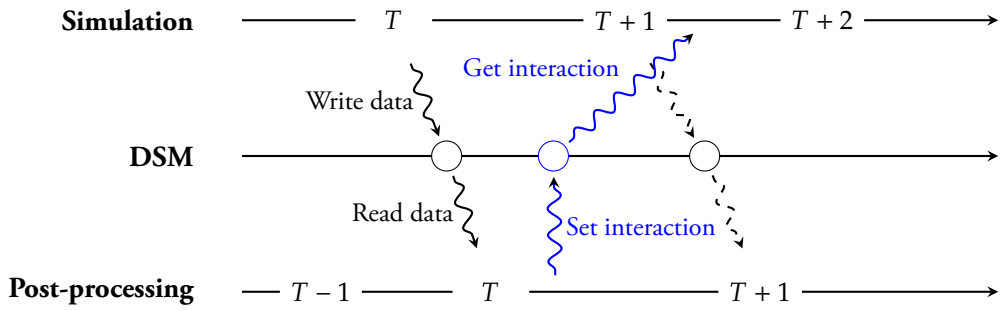


Figure 3.15. Timing of interactions in free mode.

In summary, the *wait* mode is a synchronous transmission mode of user interactions that makes the simulation stop at user defined points. The *free* mode is an asynchronous operation mode that allows the simulation to run freely, picking up interaction commands on-the-fly. 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 interactions and then revert to *free* mode again for a period. Alternatively, the switch between modes may be user-driven as a custom *command* (see chapter 4) 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.

3.4. Deployment

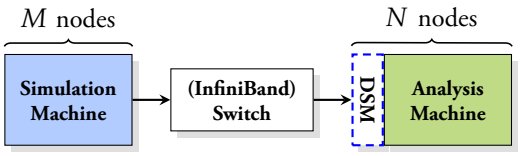
While in the previous sections we have defined a set of architectural properties and constraints, we define here different configurations available to deploy the components of our architecture. Providing the best compromises, the approach chosen will be used in the next chapter to define our implementation and the way it will be integrated.

3.4.1. Available Configurations

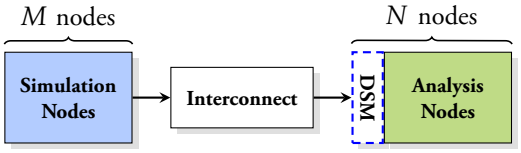
As simulation and post-processing applications interact together, the amount of time and resources allocated to compute or steer tasks has a significant impact on the overall performance of the system. For example, a simulation with very good scalability may be run on many cores using a standard amount of memory per core and efficient communications. However, the analysis required to control or steer this simulation may not scale well, or may require much more memory per node, but with a smaller number of cores. The DSM interface handles this by being quite flexible in how resources are allocated; let us consider figure 3.16, 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. It is important to note and remember that our approach is defined as a loosely-coupled approach (see 2.3) and to achieve our objective (avoid overhead on the simulation side as much as possible), the DSM component is never deployed on the simulation nodes themselves.

The first configuration (figure 3.16a) may be commonly adopted if a local cluster is considered as a simple extension of the simulation machine. 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 significant communication bottleneck. The final post-processing stage can then happen on this machine (or on another machine depending on the architecture of the post-processing application). Using separate machines makes it easy to ensure that optimized nodes (e.g., GPU accelerated nodes) are used where needed.

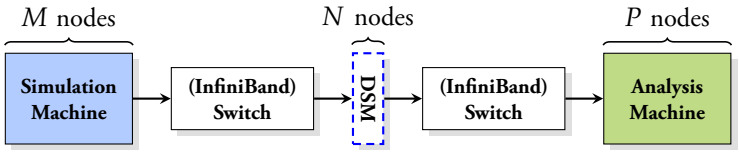
The second configuration (figure 3.16b) is more likely when large amounts of data are produced by the simulation. If a hybrid machine is available, or if the simulation and analysis make use of similar node configurations, a single machine may be used for both tasks. Note that *separate nodes* are used for the two tasks, so a judicious choice for the M and N values is still permitted so that enough DSM nodes are used to maximize the speed of the transfers and keep a good ratio between the amount of data that is generated and the number of links/processes that are used to receive data. As the two sets of nodes are linked together by using the internal system network (or



(a) Simulation and analysis on separate machines.



(b) Simulation and analysis on the same machine.



(c) Simulation on one machine, DSM on a second machine and analysis on another.

Figure 3.16. Three available configurations.

interconnect), this configuration can provide the best performance and ensure that transfers are performed at maximum speed.

The third configuration (figure 3.16c) is likely when the data generated by the simulation is smaller than in figure 3.16b and can be handled on a separate machine entirely dedicated to the analysis. While in the previous configurations DSM and analysis were tightly-coupled, in this configuration DSM and analysis stand on separate machines. The main advantage provided by this configuration is that one can dedicate a machine with a large amount of memory to host the DSM and another machine to run the analysis. While in the previous configurations, data locality could be an advantage for the analysis to reduce the amount of transfers (assuming that analysis processes need to access data from other DSM processes during a post-processing operation); in this configuration an additional $N \times P$ redistribution must be performed. This may be beneficial in some cases when data is small and analysis is well optimized on the other machine but may decrease performance in most cases because of the additional transfers and memory copies that are necessary.

Additional configurations could be created by combining for instance figures 3.16b and 3.16c. However the drawbacks and advantages previously stated would remain the same.

3.4.2. Ideal Configuration

From the three previous configurations, the one that is more likely to be used is the configuration of figure 3.16b. Again, one of our main objectives in this work is to avoid any overhead on the simulation side as much as possible, thereby transferring and staging data to other nodes for analysis and visualization. It is therefore crucial to send data to different nodes as fast as possible to avoid any bottleneck on the simulation side. Considering the simulation may run on a very large system using thousands of nodes, sending data to a different machine may not be efficient enough as external inter-connection networks are usually not as fast as the internal network that links the internal nodes together (and placing additional very high speed network switches may not be always feasible). Therefore, the best choice is to dedicate a subset of nodes (the same HPC machine used to run the simulation on) and have a staging DSM server running on these nodes.

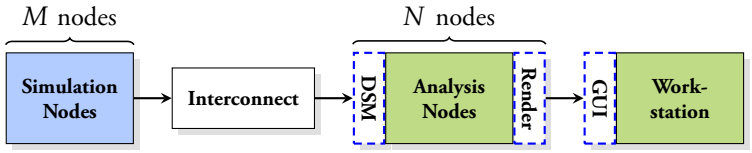


Figure 3.17. Common parallel post-processing applications use a client/server architecture.

Once this configuration set up, the post-processing application must be located *as close as possible* to the staged data to reduce extra data movement costs and run analysis algorithms. Parallel post-processing applications such as ParaView [15,70] or VisIt [19] are composed of a client/server architecture. For that reason, as shown in figure 3.17, the post-processing and analysis part of the pipeline can be represented and split into two different pieces, creating a more distributed configuration: a server acts as a host for the DSM in a multi-threaded fashion so that post-processing algorithms can take advantage of data locality; a client hosts the GUI located on a separate workstation to gather visualization results and send user interactions from/to the post-processing server.

We will focus on this type of configuration in the next section (3.5) and for our implementation in chapter 4.

3.5. Application Integration

For a better understanding of the mechanisms that are used by the different components of the architecture and to highlight the features brought by our approach, we focus in this section on a simulation example where the main object that needs to be steered is composed of a regular mesh (i.e., two different types of data arrays, point and connectivity arrays along with associated attribute

arrays that can be pressure, velocity, etc). For clarity, we focus on the configuration introduced in 3.4.2.

As we illustrated in the introduction of chapter 3, a simulation may write out data every time step (or every time step, data coming from the simulation needs to be saved). These results are saved using an hierarchical I/O interface (see 3.3.1) so that they can be written and read easily, a name being associated to each data array. At every time step, the structure of the data may follow the configuration illustrated in figure 3.18 (the same data can be organized in various ways depending on user needs).

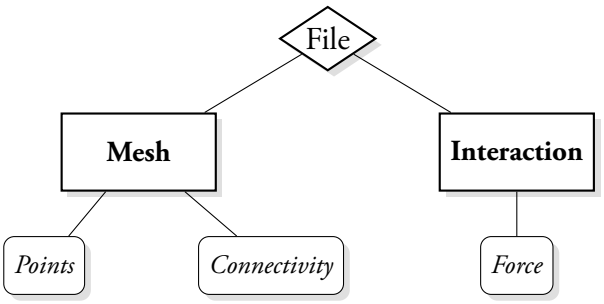


Figure 3.18. Arrays that represent the mesh are stored using an hierarchical approach (for clarity, other attribute arrays such as pressure, velocity, etc, are not represented but are stored at the same level as the point and connectivity arrays).

As previously said, the I/O interface can access the underlying DSM interface and data can be directly mapped to the DSM. Using the communication interface and synchronization mechanism introduced in 3.1, when the simulation writes data, opening the in-memory file results in locking the file from the post-processing application. Data is then sent using a two-sided or one-sided approach to the DSM processes and mapped using one of the redistribution mechanisms described in 3.2.2; depending on the size of the data that is to be sent, all the DSM processes are used to receive data. When the file is closed, using the notification system introduced in 3.1.5, a notification can be sent in a push-driven fashion to tell the post-processing side that new data has been written (more complex notifications can also be used if other types of simulation events need to be reported to the post-processing application) and the simulation can carry on computing the next simulation step.

When the post-processing application receives the notification, it opens the file and reads the data (assuming that a description of the data structure is provided to the application). In this configuration, where DSM and post-processing application share the same nodes, communication between the DSM processes and post-processing processes is performed only if data that needs to be gathered is located on a different process. When data is read, the file is closed and post-processing operations can start.

If we iterate this process over time, the user is able to analyze data in-situ while the simulation is computing. At a certain point, the user may decide to modify simulation parameters or modify the structure of the mesh that is being used by the simulation. As illustrated in figure 3.18, a parameter or a simple vector (such as a force) sent back by the post-processing application to the DSM can be stored in an *Interaction* group, which is added to the file data structure. Moreover because of the hierarchical structure of the file, modifying a mesh *only* requires modification of the point and connectivity arrays that are stored in the in-memory file, and which can be accessed through the hierarchical I/O interface. As this has been highlighted in section 3.3.4, modifying a mesh or a parameter in the simulation can be realized asynchronously or synchronously (by pausing and resuming the simulation at pre-defined points).

Once data has been written back to the DSM, data stored in the *Interaction* group can be easily retrieved from the DSM by the simulation using the higher level steering interface introduced in 3.3. When the file is re-opened to read values stored in the DSM, the same communication pattern introduced for writing data can be applied and modified mesh values can be read back using either the high level steering interface or the lower level I/O interface. All the structure changes that are read back by the simulation must be properly handled by the simulation so that no brutal change in the solver happens (which would likely crash the simulation).

As a final note, during all these operations no additional overhead (memory consumption or high-demanding CPU operation) is required to be handled at the simulation level as the various operations only require network communication.

3.6. Conclusion

In this chapter, we defined the main pillars of our architecture to create a distributed shared memory interface for the exchange of data between simulations and post-processing applications in a loosely coupled and push-driven fashion.

In 3.1, we defined a communication layer that allows transfers of in-memory file to a DSM, handling concurrent accesses and push-driven event notifications. To transfer data, we saw that a traditional two-sided interface does not allow us to take full advantage of the approach at scale, due to synchronization mechanisms and number of requests that are necessary to handle the transfers. Using a one-sided interface, data can be remotely transferred from one end to the other (assuming that the DSM remote addresses are known by the simulation), reducing the synchronization costs and number of requests as data can be sent without any participation of the DSM remote target. However, we will see in the next chapter that this communication mode can present some implementation limitations.

Once data can be sent using the communication interface previously described, we showed in 3.2 how one can optimize the use of the DSM memory space by redistributing data pieces so that all the processes that host the DSM are used to receive data and hence the network links that connect the simulation processes to the DSM. We will show in 4.1.4 the impact of these redistribution methods on the overall communication performance.

While the first two parts of this chapter were mainly focused on the efficiency of the data transfers and on the synchronization and notification mechanisms, we showed in 3.3 how we can take advantage of the existing I/O libraries that follow a hierarchical data model to easily transfer, retrieve and modify data and commands to/from the DSM. This also presupposes that the hierarchical representation of the data that is stored in the DSM is known from either side that needs to access it (simulation or post-processing). Its implementation will be further detailed in the next chapter. Using a high level interface we showed how users can be provided with a comprehensive steering interface that allows transfers and exchanges of commands and data. Moreover we showed how one can set additional synchronization points using the steering interface to create two different modes of operation, one synchronous and the other asynchronous. An implementation based on the ParaView framework will be presented in the next chapter and illustrations of this feature will be presented in chapter 5.

Finally, as the different components of our architecture can be placed and deployed in various ways, we showed in 3.4.2 that the ideal configuration consists of using a supercomputing machine with a set of nodes dedicated to the simulation and a set of nodes dedicated to hosting the DSM, which can be shared with post-processing servers so that they can access DSM data more easily. The two sets of nodes are linked together through a very high speed interconnect while a post-processing client, located on a remote machine or workstation, is connected using a standard network so that simple GUI commands can be sent to the post-processing servers.

By following this approach, we respect the two main objectives of this chapter: minimization of the overheads on the simulation side by making use of push-driven and one-sided paradigms; definition of a two-way communication model with a synchronization mechanism capable of handling concurrent accesses to the DSM.

Chapter 4.

A Parallel HDF5 Interface: Implementation and Integration

Reading from and writing to a distributed shared memory for data staging and command passing requires making use of an efficient and robust communication system, which we defined in chapter 3. In 2.3.4, we proposed to integrate our architecture directly within the hierarchical I/O library itself, so that codes that already make use of this library do not need to be reworked. In our implementation work, we decide to implement the DSM through the HDF5 library [72] (see section 4.1) and make use of the ParaView [70] framework for post-processing/GUI integration (see section 4.2). In this context, we define three main objectives: the first one is to be able to exchange data as fast as possible between the simulation and the DSM (as data movement is the potential issue of our approach, see 2.2.2); the second one is to define an interface that is portable enough to run on the widest range of systems; and the third one is (as described in 2.3.4) to minimize the required code modifications (which are necessary to be able to re-route data to the DSM), as well as proposing a generic interface that can be applied to various types of simulations.

4.1. DSM Virtual File Driver

To implement our framework, we make use of the HDF5 library [72] which is already widely adopted by the scientific community. HDF5 already provides users with several different file *drivers*. They act as an abstraction layer between the high level HDF5 API and the lower level I/O protocols. As shown in figure 4.1, the drivers provided by the HDF5 package include essentially the `sec2` (POSIX compliant serial I/O), `core` (memory based), and `mpio` (MPI I/O) drivers. In this section, we present the implementation of a `dsm` driver.

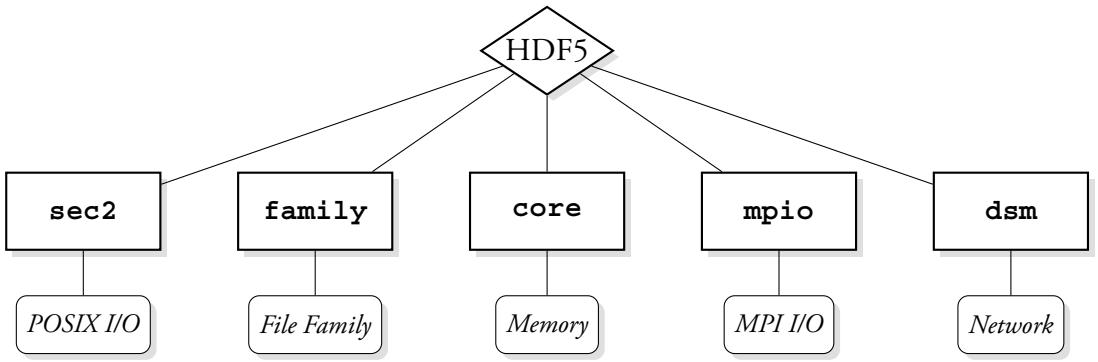


Figure 4.1. The HDF5 library defines a virtual file layer allowing users to switch between different I/O methods (e.g., MPI I/O, POSIX I/O, etc).

4.1.1. Driver Implementation

The HDF5 virtual file layer follows a modular architecture and it is worth mentioning that additional virtual file drivers have already been developed such as the `stream` driver [5], which has been created to provide live access to simulation data by transfer to remote grid servers or via sockets to a waiting application. Whereas this driver could be used for data staging purposes, it is unfortunately a serial driver (as opposed to a parallel driver) and only allows serial transfers of data to the memory of a remote application.

The original DSM implementation (upon which this work is based), referred to as the *Network Distributed Global Memory* (NDGM), was created by ARL [20, 22]; it was used for code coupling between CFD applications, modeling fluid-structure interactions [23] using very different models (and hence partitioning schemes). Separate codes may write their data using any HDF5 structures suitable for the representation, providing the other coupled processes are able to understand the data and read it with their own partitioning scheme. The original NDGM implementation supported the transfer of data between processes using only a single channel (as the `stream` driver does) to a DSM distributed among processes of a tightly coupled application and therefore had a limited capacity. Taking it as a basis, we enhanced this driver to support parallel data transfers between loosely coupled applications (that effectively stand on different resource partitions as opposed to the original implementation), creating a new driver called the `dsm` driver.

The `dsm` driver that we implement is a parallel driver (i.e., all the internal intra-communication is based on MPI transfers; this is imposed by the HDF5 virtual file layer). For inter-communication and remote exchanges, we define different communicator types, two-sided and one-sided, as described in section 3.1. We detail the implementation of these communicators in 4.1.3.

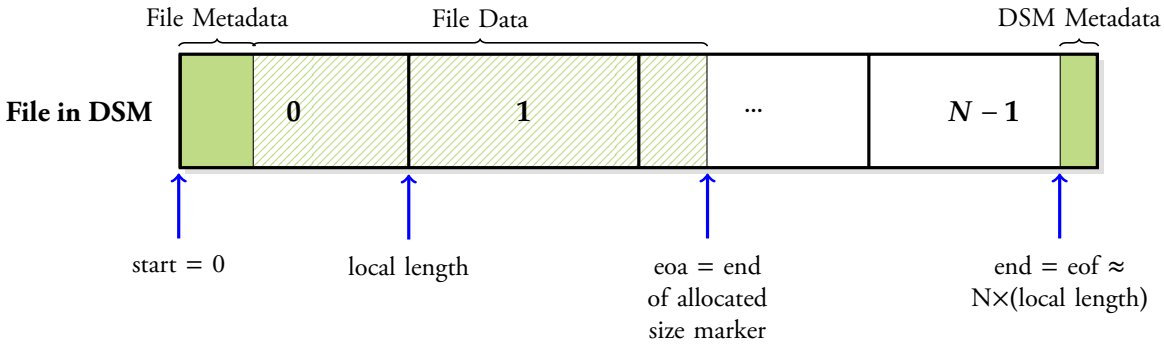


Figure 4.2. Every DSM file can be seen as a contiguous memory space that can be filled using different redistribution patterns. For readability, we represent here a uniform redistribution: HDF5 metadata is stored along with data and DSM metadata is stored in a reserved section of the memory space.

As described in figure 4.2, the DSM memory space can be seen as a contiguously distributed memory space. Each of the DSM processes effectively allocates a buffer of size *local length*; the DSM is filled from rank 0 to rank $N - 1$ as the library writes data pieces contiguously (by default) but may use different redistribution strategies as explained in 3.2.2. Whereas all the HDF5 metadata (see figure 2.2) is stored and managed in the memory space reserved for the HDF5 file itself, we additionally reserve a section at the end of the file to store DSM metadata. This metadata contains the *start* and *end* of allocation (*eoa*) addresses of the memory mapped file (which are described in figure 4.2) as well as potential steering commands or steering information that do not need to be seen in the file (from a user point of view, e.g., pause, etc).

4.1.2. Driver Usage and Restrictions

When writing in parallel using the HDF5 API, it is necessary to select a parallel file driver from within the application code. The only one currently available is the `mpio` driver, which would normally be selected by setting a file access property list using the function:

```
herr_t H5Pset_fapl_mpio(hid_t fapl_id, MPI_Comm comm, MPI_Info info);
```

```
h5pset_fapl_mpio_f(prp_id, comm, info, hdferr)
        INTEGER(HID_T) prp_id
        INTEGER comm, info, hdferr
```

To use the `dsm` driver from C or Fortran, one may change the previous lines with the corresponding `dsm` driver calls:

```
herr_t H5Pset_fapl_dsm(hid_t fapl_id, MPI_Comm intra_comm, void *
    local_buf_ptr, size_t local_buf_len);
```

```
h5pset_fapl_dsm_f(prp_id, intra_comm, hdferr)
        INTEGER(HID_T) prp_id
        INTEGER intra_comm, hdferr
```

Doing this effectively modifies the virtual file driver used by the HDF5 library so that all the following calls redirect data to the DSM. We define two operation modes: *client* and *server* modes. In a common scenario (which also corresponds to the configuration of section 3.4.2), a simulation that wants to offload its data to a remote DSM operates as a client while the post-processing application hosting the DSM server reads back from it. In client mode, it is worth noting that setting the `local_buf_ptr` variable to `NULL` or using the Fortran API instructs the library to operate on a remote DSM buffer (so no memory buffer is locally allocated in this mode); this is illustrated in the following example (which is also what one would typically do in its simulation code):

```
int main()
{
  ...
  /* HDF5 calls that follow H5Pset_fapl_dsm() use the dsm driver. */
  fapl_id = H5Pcreate(H5P_FILE_ACCESS);
  H5Pset_fapl_dsm(fapl_id, MPI_COMM_WORLD, NULL, 0);
  /* Create HDF5 file/groups/datasets collectively. */
  file_id = H5Fcreate(filename, H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
  H5Pclose(fapl_id);
  file_space_id = H5Screate_simple(rank, count, NULL);
  dataset_id = H5Dcreate(file_id, "dataset_name", H5T_NATIVE_DOUBLE,
        file_space_id, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
  mem_space_id = H5Screate_simple(rank, localdim, NULL);
  /* Writes can be independent. */
  H5Sselect_hyperslab(file_space_id, H5S_SELECT_SET, offset,
        NULL, localdim, NULL);
  H5Dwrite(dataset_id, H5T_NATIVE_DOUBLE, mem_space_id, file_space_id,
        H5P_DEFAULT, data);
  /* Release IDs. */
  H5Sclose(mem_space_id);
  H5Dclose(dataset_id);
  H5Sclose(file_space_id);
  H5Fclose(file_id);
  ...
}
```

In server mode, setting this same variable to `NULL` instructs the library to auto-allocate and manage a singleton DSM for the user. For both modes, the communicator `intra_comm` is used as the MPI intra-communicator. Note also that in server mode we define two ways of initializing a DSM buffer: either by supplying a pointer to a memory buffer that has been previously allocated, or (for advanced controls) by supplying a pointer to a C++ class object called `H5FDdsmManager` (see annex B).

HDF5 Restrictions

When writing using a parallel virtual file driver, HDF5 imposes all the metadata operations to be collective. As illustrated in figure 4.3, one must therefore create, open the in-memory file collectively and create a new group or dataset collectively as well before being able to issue independent write operations. This may be a potential restriction for simulation code developers as it requires all the processes to collectively gather data information (size of data chunks, etc) before being able to create HDF5 data structures (collectively). The HDF5 library requires DSM metadata information (i.e., the *eoa* size marker of figure 4.2) to be synchronized so that two different processes do not write to the same memory space. As this is more an HDF5 issue than a `dsm` driver issue, this point will not be further discussed here, but one of the solutions that is going to be implemented in the HDF5 library is the definition of dedicated server processes for the collection of metadata.

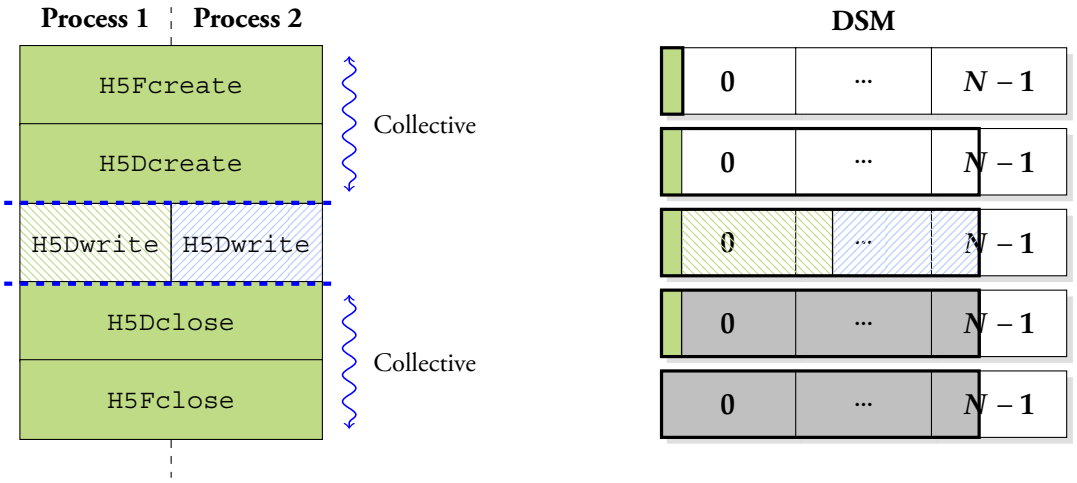


Figure 4.3. When using a parallel virtual file driver, HDF5 requires metadata operations to be issued collectively. For clarity, only a contiguous data distribution is represented.

Nonetheless if the in-memory file is opened in read-only mode, metadata will not be subject to any modifications and collective metadata synchronization steps can therefore be skipped. When

write access is given, these synchronization steps are necessary again so that as soon as the file structure is modified, all the processes keep their metadata (and hence *eoa* size marker) synchronized.

4.1.3. Platform Optimization and Inter-Communicators

To write and transfer data from the simulation to the DSM, we can define and implement different inter-communicators, two-sided and one-sided (see 3.1). While the fundamental difference between these two categories is the mode of transfer and the synchronization algorithm, we also distinguish two different forms of communicator creation: *static* and *dynamic*. The *static* communicator creation requires the two applications to be launched within the same job whereas the *dynamic* creation is more flexible and (if the system supports it) allows the applications to be connected/disconnected dynamically at the cost of additional handshakings. We present three different approaches: one that uses POSIX sockets and allows heterogeneous systems to be connected together, one based on MPI and MPI Remote Memory Access[1] (RMA) [2,33] that provides better performance as it uses network specific communication layers underneath but presents API limitations, and one based on a Cray API called DMAPP [13] that offers high throughput but is limited to Cray systems.

Systems used for testing. For testing and comparison we will make use of two systems: an InfiniBand QDR 4X cluster with MVAPICH2 1.8 [73] composed of 12 core AMD Magny-Cours (i.e., 12 nodes, 144 cores) and a Cray XK6 system composed of two 16-core AMD Interlagos cabinets (i.e., 176 compute nodes, 2816 cores) with Cray MPT 5.4 (derived from MPICH2 [7,62]), which features a Gemini interconnect [6]. For reference, the theoretical inter-node bandwidth of the InfiniBand cluster is 4 GB/s. For the Gemini interconnect used in the Cray XK6, as different transfer methods can be selected (see annex A), we take for reference the maximum bandwidth reached for large messages using the RDMA (Remote Direct Memory Access) BTE (Block Transfer Engine, see annex A) mechanism, 7 GB/s.

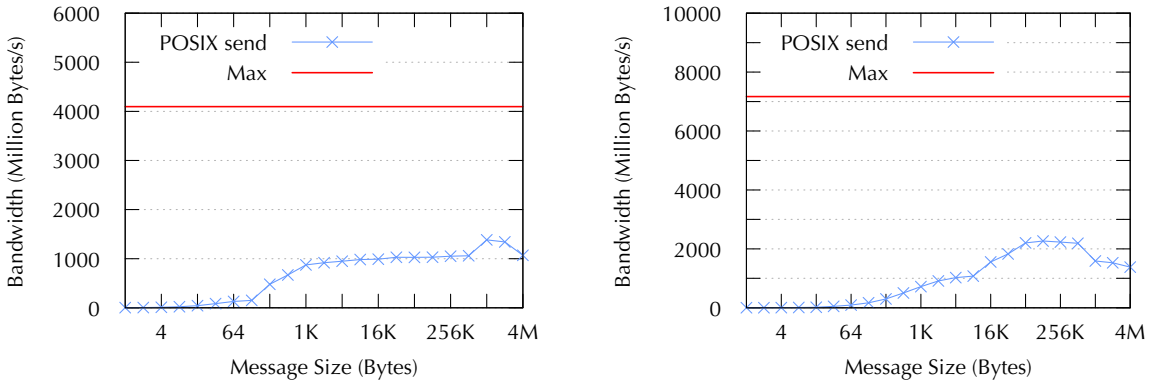
Socket

To allow our driver to be used on heterogeneous systems, we have introduced a socket inter-communicator. It uses a single socket to initialize the connection (*dynamically*) between both applications, then creates additional sockets to link every node of one application to every node of

[1] Similar to the concept of one-sided communication defined in 3.1.3. Note that the term *RDMA* (Remote Direct Memory Access) is not explicitly used here, as the underlying MPI implementation may involve the operating system to perform the transfers.

the other. As this is a two-sided approach, transfers are realized using POSIX `send/recv`. Many operating systems currently limit the number of open socket connections to around 1024, placing a (configurable) limit on the number of descriptors that can be maintained at any given time. In the case where this limit can be configured, the array of file descriptors that needs to be stored grows as much as the number of processes increases. To handle this problem, a solution would be to manage connections created on-the-fly but this implies an additional maintenance cost. Another solution could also be to make use of hierarchical communication schemes, performing communications only between smaller subsets of processes.

The main advantage currently given by this inter-communicator is that it does not depend on the MPI implementation used within the connected applications; therefore it is possible to create connections between any combination of clusters or machines. Although this solution is *not* a scalable solution, it does provide a reliable way of coupling heterogeneous systems.



(a) InfiniBand QDR 4X cluster.

(b) Cray XK6 system.

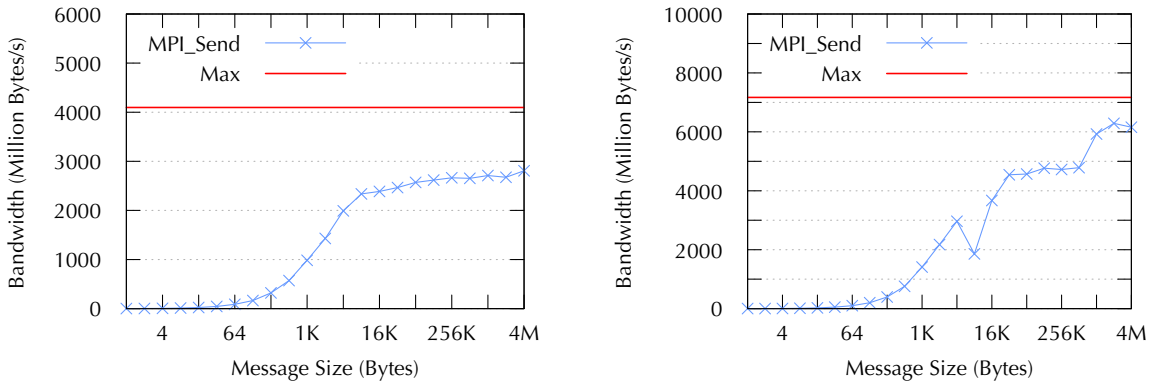
Figure 4.4. Inter-node micro-benchmark using POSIX socket transfers.

Using the system internal network, one may consider to use sockets as well, but as shown in figure 4.4, the point-to-point bandwidth obtained between two nodes is much lower than the theoretical network bandwidth. For this reason (see 3.4.2), we will not study this communicator further and focus on the other communicator types that take advantage of system specific communication layers.

MPI and MPI RMA

The MPI inter-communicator is intended to be used when the simulation code and the DSM are on the same machine, or on machines that have compatible MPI process managers. To establish the connection dynamically between the applications, we make use of the `MPI_Comm_con-`

nect and MPI_Comm_accept function calls. Unfortunately, some large machines such as Cray XT/XE/XK or IBM Blue Gene systems are unable to use the dynamic process management set of functions and hence the MPI_Comm_connect and MPI_Comm_accept functions (which is mainly due to limitations in the resource allocation systems). In this case the two applications must be launched as part of the same job (MPMD) and the global communicator MPI_COMM_WORLD must be split between the applications using an MPI_Comm_split call. Note that this forbids the use of MPI_COMM_WORLD within the applications. Once the global communicator split, a call to MPI_Intercomm_create creates an inter-communicator to communicate between the two applications.



(a) InfiniBand QDR 4X cluster.

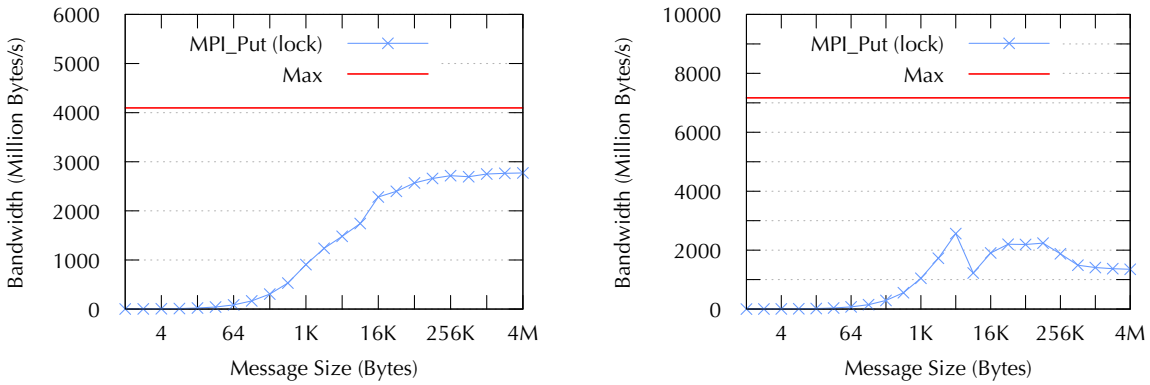
(b) Cray XK6 system.

Figure 4.5. Inter-node micro-benchmark using point-to-point MPI transfers.

For reference, the bandwidth between two nodes using MPI point-to-point communication is represented in figure 4.5. On both systems, one may note that single MPI_Send calls present good performance as they take advantage of RDMA (Remote Direct Memory Access), bypassing the OS for direct access to the remote memory. On the Cray XK6, one may note a performance drop point for message sizes of 8 kB (this limit may vary with the number of processes in the job). This point corresponds to the standard offload thresholds, making use of the block transfer engine for large messages (more details on the different offload mechanisms are presented in annex A).

MPI one-sided communication has been introduced in MPI 2 and is particularly appropriate to situations where the remote memory target is already known by the source processes, avoiding extra and unnecessary point-to-point operations to be issued (see section 3.1.3). The MPI RMA communicator that we define makes use of the passive MPI RMA synchronization mechanism, which means that no active participation of the remote target is requested (or necessary) between distinct access epochs. When the DSM is allocated, we make a call to MPI_Alloc_mem, which effectively allows the library to make memory optimization for further accesses. We then define a window, set as the size of the requested HDF5 file. MPI_Put can then be issued in a one sided

manner using `MPI_Win_lock` and `MPI_Win_unlock` between transactions; as defined in the MPI specification, transactions complete when the window is unlocked. In figure 4.6 that shows simple transactions between two nodes, note that the MPI one-sided interface on the Cray XK6 does not perform as well as expected; the implementation (which is based on MPICH2) does not support large message transfer (LMT) optimization for one-sided protocols and this issue should be corrected with future RMA implementations that follow the MPI 3 specification [32].



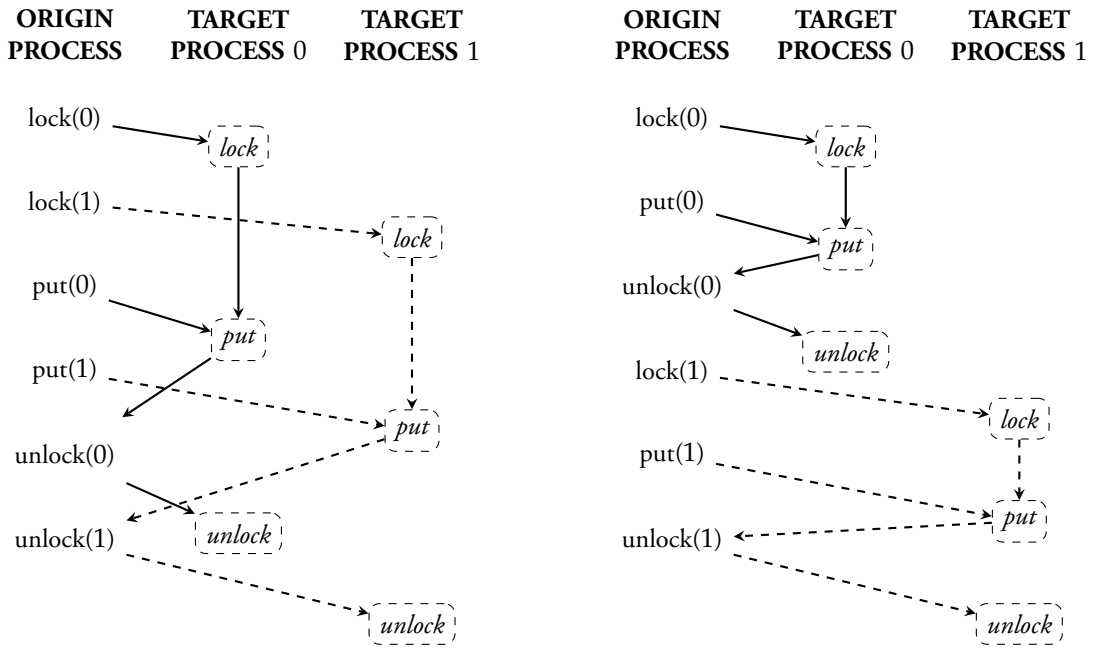
(a) InfiniBand QDR 4X cluster.

(b) Cray XK6 system.

Figure 4.6. Inter-node micro-benchmark using passive MPI RMA transfers.

While this approach may seem to be the best approach [34], the current MPI 2 API presents serious restrictions [10]. One of them is illustrated in figure 4.7. When data is sent to the DSM from a source process, it is important to be able to perform multiple put operations to multiple target processes (particularly so when using a block-cyclic redistribution) though the MPI 2 standard does not allow this type of operation, and RMA operations on a given window can only access the memory of a single process during an access epoch. This is an important restriction for our DSM approach, as data mapped (contiguously or following a certain redistribution pattern) to the entire DSM forces the simulation processes to access the local memory of multiple target processes. This therefore leads us to serialize transfers, locking and unlocking remote targets between memory accesses (as described in figure 4.7b). As an alternative, one may wish to create memory windows dynamically or separate memory windows mapped onto distinct processes. However window creation is a collective operation where the source and target processes that intend to use that window must be included. This solution is therefore complex to handle and fortunately the upcoming MPI 3 specification will address these issues by providing dynamic window creation and multiple target locking.

DSM single dataset benchmark. Using a contiguous data redistribution, bandwidth results are presented in figure 4.8. In this benchmark, write bandwidth tests can be seen as typical clien-

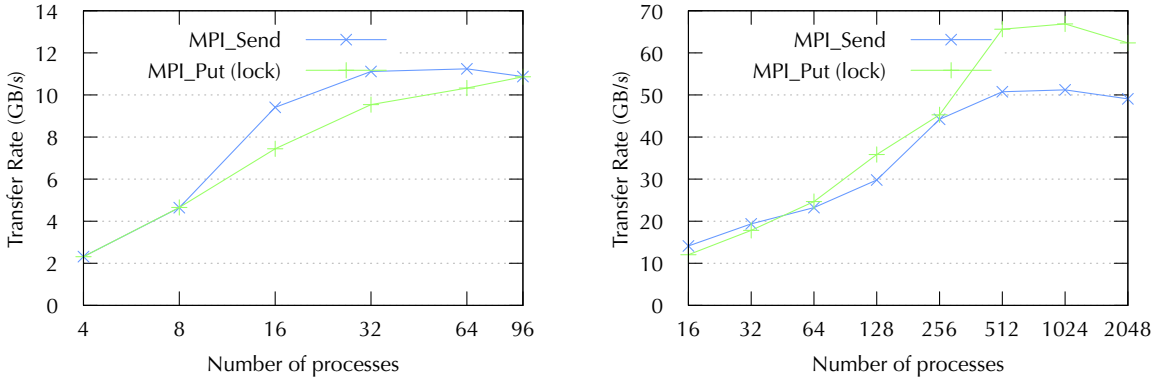


(a) Multiple targets are concurrently accessed. **(b)** Multiple targets are serially accessed.

Figure 4.7. One of the passive MPI synchronization restrictions is the inability of locking multiple target processes of the same window per access epoch.

t/server tests: a first set of processes (servers) hosts the DSM and waits for incoming data, a second set of processes (clients) writes HDF5 data in parallel to the DSM using the `dsm` driver. The aggregate bandwidth that we measure corresponds to the amount of data sent during a complete file write (HDF5 create, write and close operations). The DSM is distributed among 4 nodes (16 processes, 4 processes per node) on the InfiniBand cluster and among 40 nodes (160 processes, 4 processes per node) on the XK6. To keep a certain consistency between the systems, the local buffer size allocated per node is kept to 1 GB, which creates a DSM of 4 GB on the InfiniBand cluster and a DSM of 40 GB on the XK6. Given this fixed DSM (file) size, a single dataset of the matching size is written from the combined send nodes (smaller pieces per process as the number of processes increases, i.e., each process sends pieces of the DSM size divided by the total number of processes sending data).

For writing, the number of processes on the XK6 is 4 per send node until 128 nodes are used (512 processes), at which point to saturate the bandwidth the number of processes per send node is increased up to 16 (giving 2048 processes writing data in total). On the InfiniBand cluster, 8 send nodes are available and 4 processes per node are used initially and then incremented to 12 per send node to saturate the bandwidth giving a maximum of 96 send processes. As we are limited here by the number of network adapters and not by the number of processes per node, keeping



(a) InfiniBand QDR 4X cluster—DSM size of 4 GB and distributed among 16 processes (4 nodes).

(b) Cray XK6 system—DSM size of 40 GB and distributed among 160 processes (40 nodes).

Figure 4.8. Write transfer rate of an (in-memory) HDF5 file composed of one single dataset using a contiguous distribution—Comparison between MPI point-to-point and passive RMA transfers.

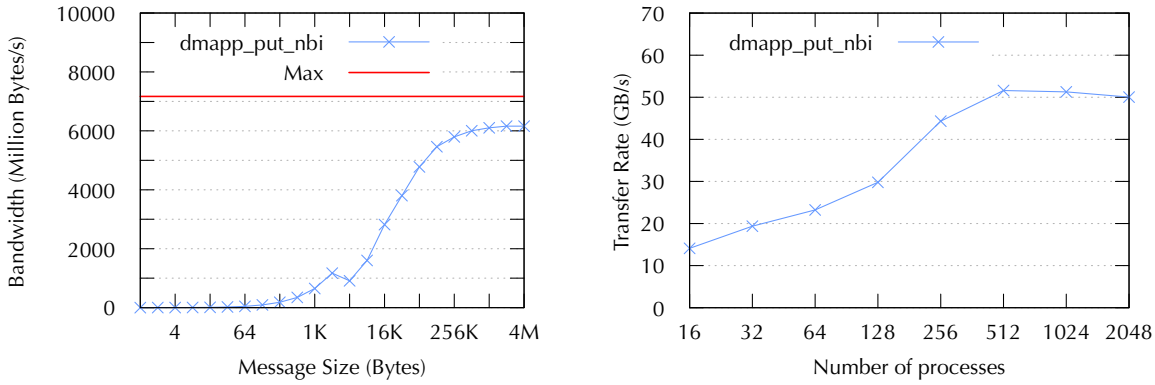
initially a low number of processes per node allows us to reach higher bandwidth results. The effect of bandwidth saturation is then shown when the number of processes per node is maxed out. This configuration may reflect a typical usage of the `dsm` driver where a small partition of nodes is dedicated to hosting the DSM and post-processing the data while a larger partition is dedicated to running the simulation.

As shown in figure 4.8, for smaller number of processes, the performance reached using two-sided communication is better than the one reached using one-sided communication (as point-to-point communications have been highly tweaked in MPI implementations). However when reaching larger number of processes, one-sided communication provides better performance. As we described in 3.1.2 and in 3.1.3, the synchronization algorithm used for the two-sided approach as well as the number of requests that need to be issued to transfer data makes this two-sided communicator less scalable than the one-sided communicator. This behavior is more visible on the Cray XK6 where a larger number of processes can be easily reached.

DMAPP

The DMAPP communicator is derived from the aforementioned MPI RMA communicator. On Cray XE/XK systems that support the latest generation of interconnect Gemini [6], Cray defines the Distributed Memory Application API, referred as DMAPP [13]. This API is used on these systems to implement one-sided libraries such as Cray SHMEM [25] and is also used by PGAS [1] compilers (Co-array Fortran and UPC).

We have implemented a communicator that takes direct advantage of this lower level one-sided communication library. On the simulation side, to avoid memory overheads created by symmetric memory usage (see annex A), we allocate memory from the private heap and register it to the DMAPP API on the DSM hosts only. This registration step provides memory segment information, which is then exchanged with the simulation (only once at initialization time, assuming that the DSM size is fixed between time steps), as shown in 3.1.3. `dmapp_put` calls can then be issued to transfer data into the DSM, the *Memory Relocation Table* (MRT) on the Gemini NIC mapping the memory references contained in the incoming network packets to the physical memory on the local node. `dmapp_put` calls are blocking calls, but DMAPP provides users with a smarter mechanism called *non-blocking implicit* put operations (`dmapp_put_nbi`). This allows non-blocking put to be issued to any remote target, as memory access concurrency is here handled at the DSM library level. When the DSM file is closed and the file lock is released, all the remote transfers must complete (which is handled by `g_sync_wait`). This mechanism is particularly useful when several small messages are issued to different remote targets as the bandwidth is used more efficiently. While we lose here portability, the DMAPP communicator allows us to work around the previously described MPI RMA restrictions.



(a) Inter-node micro-benchmark using DMAPP put operations (blocking) with an FMA/BTE switch at 4 kB, which effectively corresponds to a bandwidth drop point.

(b) Write transfer rate of an (in-memory) HDF5 file composed of one single dataset using a contiguous distribution—DSM size of 40 GB and distributed among 160 processes (40 nodes).

Figure 4.9. DMAPP communicator benchmark on a Cray XK6 system.

As illustrated in figure 4.9a, DMAPP enables optimization of one-sided transfers. In figure 4.9b, we make use of the same configuration described in the previous section. When writing a single large dataset using a contiguous distribution, one may however notice that the performance does not reach the one obtained using MPI RMA (because of internal optimization within the MPI library itself). However the DMAPP API still provides in this case a scalable solution and

will show its strength when writing multiple datasets, which corresponds to a more realistic test case as we will see in the next section.

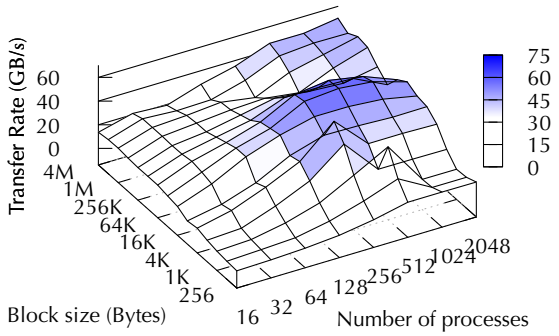
4.1.4. Impact of Redistribution Strategies

In the previous section, we showed how the different communicator types that we implemented allow us to optimize transfers depending on the system that is used and depending on the number of processes that write to the DSM. We also showed that one-sided communicators were a more scalable solution compared to a two-sided approach. However as we showed in 3.2, having data written in parallel to a contiguous memory space does not allow us to make use of all the receiving nodes, particularly so if several small datasets are written. In this section, we show how one can combine different communicator systems and redistribution strategies to improve the efficiency of the communications, which will guarantee that data exchanges made in parallel are realized as fast as possible. Therefore we only focus in this section on *one-sided* communicators.

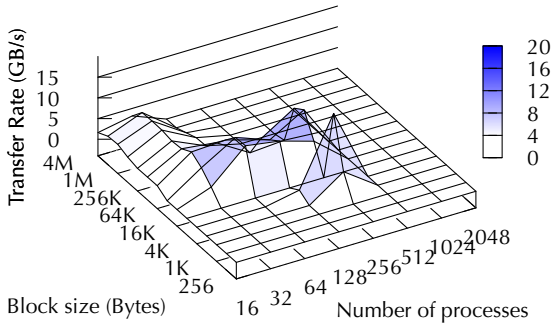
Block Redistribution

In a first time, using a block-cyclic redistribution, we want to evaluate the block sizes that can allow transfers to be maximized. For different block sizes, we run the same benchmark as the one used in the previous sections, using a single large dataset. For clarity, we only show results on the Cray XK6 system in figures 4.10 and 4.11 (additional results are presented in [84]). As a reminder, for writing, the number of processes on the XK6 is 4 per send node until 128 nodes are reached (512 processes), at which point to saturate the bandwidth the number of processes per send node is increased up to 16 (giving 2048 processes writing data in total). Bandwidth drop points highlighted in the previous sections can be observed in these figures: at 8 kB and above 256 kB for the MPI one-sided communicator, and at 4 kB for the DMAPP communicator (only visible in this figure after a careful examination). While these drop points had a small effect on the micro-benchmark, they can lead to a significant slow-down in the HDF5 write operations when those block sizes are used repeatedly.

As one may notice from figures 4.10b and 4.11b, writing a single large dataset using a block-cyclic redistribution is not beneficial when several processes are involved and when (like in this case) the number of nodes used to send data is larger than the number of receiving nodes. However, on this XK6 system, a significant improvement compared to the contiguous write is noticeable for block sizes belonging to the [16 kB, 64 kB] interval with the MPI RMA communicator and for block sizes below 4 kB for the DMAPP communicator. We will re-use these block sizes in the next sections as they obviously provide the best performance. Random block results are not shown

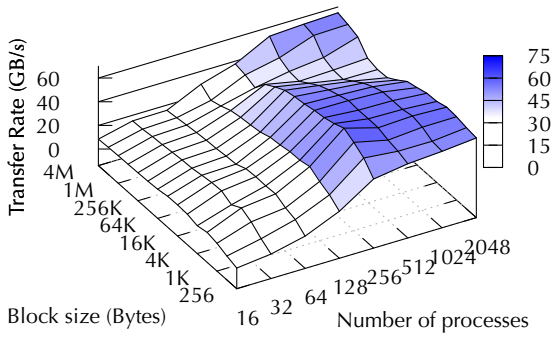


(a) Block-cyclic redistribution.

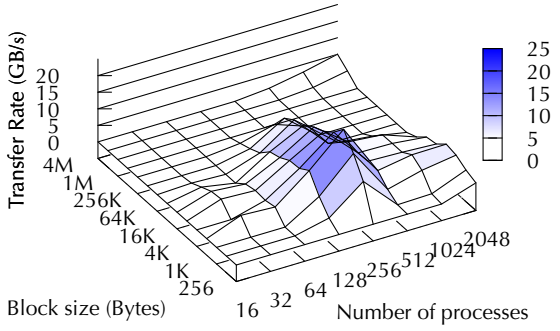


(b) Comparison between block-cyclic and contiguous distributions (difference).

Figure 4.10. Write transfer rate on Cray XK6 of an (in-memory) HDF5 file composed of one single dataset using MPI passive RMA transfers—DSM size of 40 GB and distributed among 160 processes (40 nodes).



(a) Block-cyclic redistribution.



(b) Comparison between block-cyclic and contiguous distributions (difference).

Figure 4.11. Write transfer rate on Cray XK6 of an (in-memory) HDF5 file composed of one single dataset using non-blocking implicit DMAPP transfers—DSM size of 40 GB and distributed among 160 processes (40 nodes).

here for clarity, but can globally increase the bandwidth and avoid possible congestion issues in the DSM.

Writing Multiple Datasets

We showed in 3.2.1 that when using a contiguous redistribution, writing multiple datasets results in distinct transfers through a partition of the DSM links that are available (whereas writing a single large dataset always makes use of all the DSM links at the same time). To reflect the behavior of a common simulation code, the previous benchmark is reused here, this time creating a file composed of 10 datasets instead of a single one. The same configuration is used as the previous tests; each of the datasets has the same fixed size and their sum is the size of the allocated DSM. Results are shown in figure 4.12.

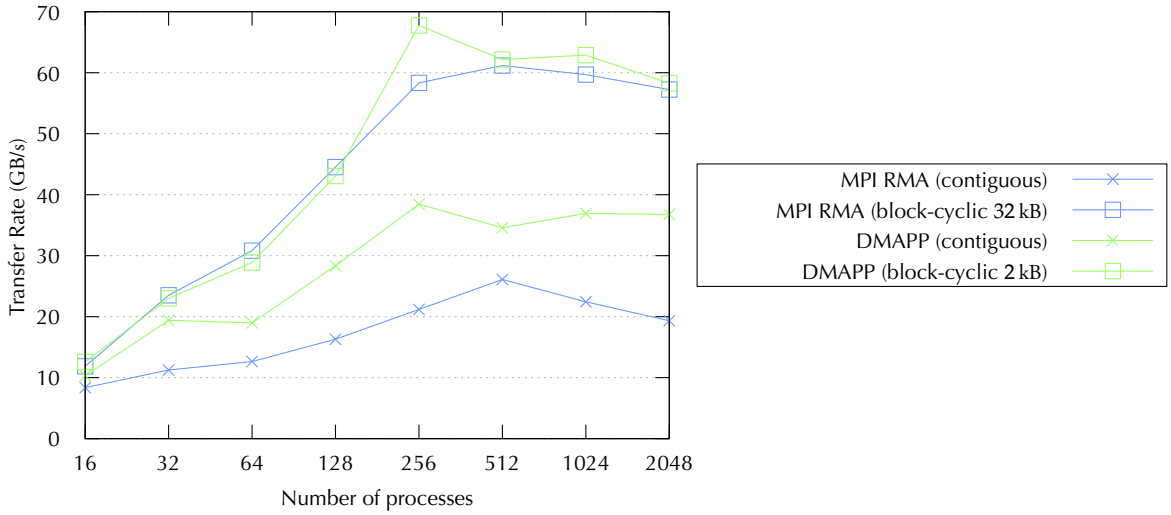


Figure 4.12. Write transfer rate on Cray XK6 using MPI RMA and DMAPP communicators of an (in-memory) HDF5 file composed of 10 datasets—DSM size of 40 GB and distributed among 160 processes (40 nodes).

Before commenting on the real advantage that the block-cyclic redistribution offers here, it is worth noting the real gain that the DMAPP interface provides with compared to MPI RMA, even on a contiguous redistribution (when writing multiple datasets). Because of the flexibility of the DMAPP API, datasets can be asynchronously sent to different remote targets as opposed to the MPI RMA implementation where transfers are serialized due to consecutive lock/unlock operations. This therefore leads to a more even use of the remote links, which concretely increases the bandwidth.

Using previously determined block sizes that improve the write performance (in this case 32 kB for MPI RMA and 2 kB for DMAPP), one can see from figure 4.12 that writing using a block-cyclic redistribution is much more efficient than linear mapping. Since each dataset in the linear HDF5 memory space is contiguous, writing 10 datasets in parallel but sequentially in time, causes only one tenth of the links to become active for each individual dataset. By redistributing blocks for each of the much smaller datasets across all processes, we make use of all of the links for all of the transfers. Whereas the performance reached using DMAPP was considerably better than MPI RMA using a contiguous redistribution, when using a block-cyclic redistribution, results between DMAPP and MPI RMA are much closer. DMAPP even in this case performs slightly better and reaches 68 GB/s for 256 processes (64 sending nodes), point where the number of sending nodes is close to the number of receiving nodes.

Reading Multiple Datasets

In the ideal configuration of section 3.4.2, the simulation writes to the DSM through the network and the post-processing application reads back from it. It is important to note that in this configuration the DSM and the post-processing application share the same nodes. To understand what the resulting bandwidth can be when reading back from the DSM to the post-processing application, we make use of the same configuration as for writing and replace senders by receivers. The measured bandwidth is the bandwidth of processes that are co-located with the DSM and read from the DSM hosts. Note that in this case, a higher bandwidth than the one we can get for write operations can be reached as depending on data location, getting data from the DSM is either a local memory copy from the DSM to the host memory (e.g., of the post-processing application) or a get operation from a remote DSM node. Note also that only blocking operations are used here as when reading the in-memory HDF5 file, HDF5 structures must be created on-the-fly and therefore need to have access to the data at the time they are actually created[2]. For block-cyclic redistribution, we make use of the same block sizes that are used for writing data (i.e., 32 kB for MPI RMA and 2 kB for DMAPP) as making use of a given size of block for writing imposes the same size of block to be used for reading (so that one can retrieve data from the DSM, see 3.2.2).

Results are shown in figure 4.13. As expected, using a block-cyclic redistribution is here again more efficient and allows us to almost double the transfer bandwidth on large number of processes. While using MPI RMA seems to be more efficient, the bandwidth starts to flatten for a large number of processes whereas DMAPP continues to scale, overtaking MPI RMA. This can be explained by very low latency of DMAPP operations and lack of optimization in the MPI one-

[2]Another solution could also be to make non-blocking operations block earlier when needed.

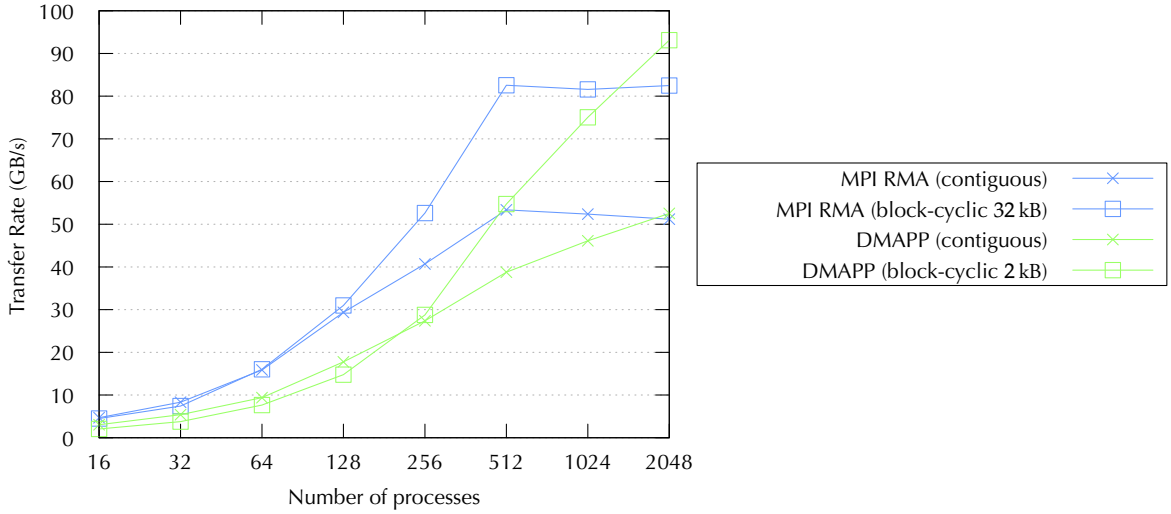


Figure 4.13. Read transfer rate on Cray XK6 using MPI RMA and DMAPP communicators of an (in-memory) HDF5 file composed of 10 datasets using a DSM size of 40 GB—160 send processes (40 nodes).

sided implementation. Using an appropriate post-processing reader, we will see in section 4.2.2 what the actual and corresponding read performance one can get within ParaView.

4.1.5. Implementation Conclusions

Implementing a distributed shared memory interface within the HDF5 library offers multiple advantages. As we showed in 4.1.2, codes that already make use of the HDF5 interface only need a single line of code to be changed to make use of our driver and see the data being re-routed in parallel to the staging area. The in-memory file stored in the DSM can then be accessed as a normal file from the user point of view and all the existing HDF5 capabilities can be re-used to read from and write to the DSM. It is worth mentioning that only one file can be stored for now in the DSM, but there is no restriction for implementing a more advanced mechanism capable of managing multiple files, giving more flexibility to simulations that write data to different destinations. An alternative could also be to store and even mount (see `H5Fmount`) multiple files into separate HDF5 groups, simulating a real file system.

The mapping between HDF5 memory objects and the actual DSM storage can be highly tuned so that one can optimize the redistribution on a given system configuration. In a common scenario, it is assumed that the simulation will write files composed of multiple datasets and being able to take advantage of all the DSM links for every write operation is an imperative requirement for our

interface. Using specific communication protocols such as DMAPP allows the current restriction imposed by MPI passive RMA to be bypassed and the DSM interface can therefore achieve high speed transfers, reducing the main overhead introduced by the approach as all the data (or a subset) must be copied from the simulation to the staging area (i.e., here the DSM).

In the next section, we show how this interface is integrated into the well known visualization and post-processing application, ParaView.

4.2. ICARUS ParaView Plug-in

While any HDF5 based applications may be coupled together with an implied assumption that one will be the master and the others the slaves, we describe in this section the enhancements that we have made to the ParaView package to allow flexible creation of a customized visualization and 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.

4.2.1. ParaView Client/Server Architecture

ParaView follows a client/server architecture and mainly supports two different modes: a *built-in* mode where all the components stand on the same machine and a *client/server* mode where components can be distributed among multiple machines.

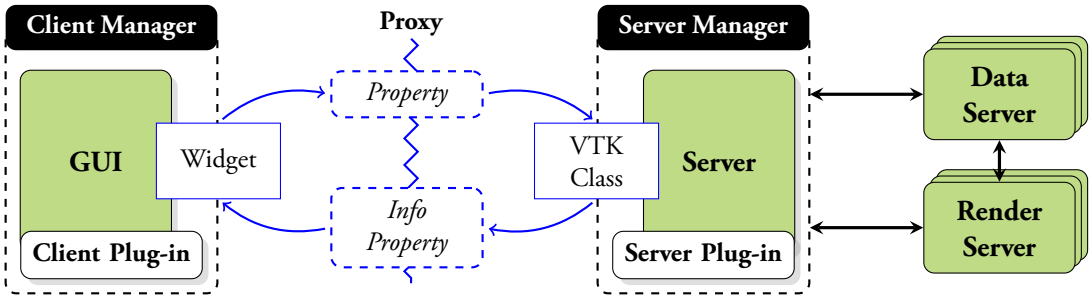


Figure 4.14. ParaView follows a client/server architecture where client and server communicate through a proxy interface.

As described in figure 4.14, in client/server mode, the GUI client part connects (using sockets) to a server part, which executes the different processing steps of the visualization pipeline (see section 2.1). The server can be itself decomposed into *data* and *render* servers to distribute and optimize the various stages of the post-processing workflow. Images or data are then transmitted

to the GUI client using a *proxy* layer, which manages the message flow between the server and the client. When sending a command from the GUI to the server, the remote ParaView server instances are updated so that data and orders are received and processed. The proxy layer mechanism and server manager properties (*property* and *info property* in figure 4.14) ensure that information and data between a given GUI widget (e.g., GUI button, check box, etc) and the corresponding VTK class server object are correctly transmitted and synchronized.

The ParaView architecture is very modular and new functionality can be added through a plug-in mechanism. Practically, creation of a plug-in requires definition of the two client and server parts; the *property* and *info property* defined will then transmit commands and data between the GUI controls and the VTK objects created in the plug-in. To take advantage of the ParaView architecture and define a generic user interface that can be used with any type of simulation, the ICARUS plug-in, which must be the interface between the simulation and the ParaView framework, relies on the HDF5 library and `dsm` driver defined in 4.1. Whenever a command is queried in the GUI, the associated event is forwarded to the ParaView server through the proxy communication layer, which transmits it in turn to the defined VTK objects and to the `dsm` driver library.

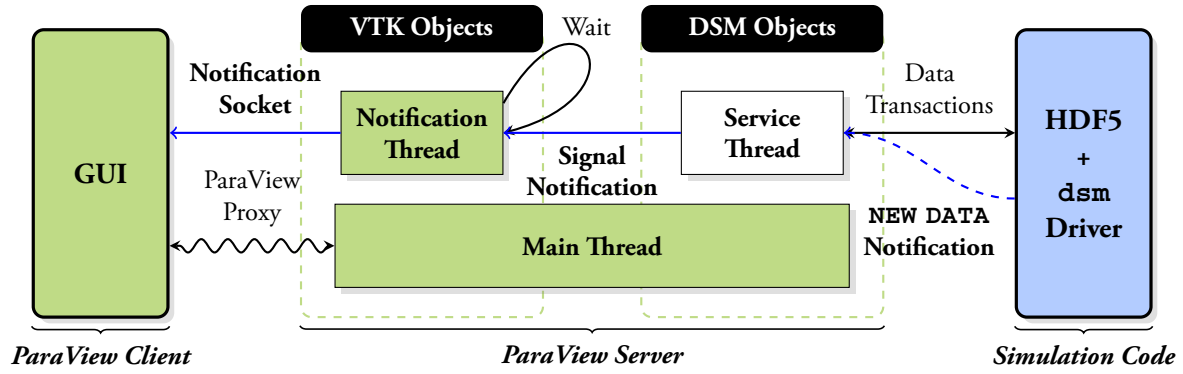


Figure 4.15. Using the notification mechanism described in 3.1.5 with the ParaView framework, when a new data or new information notification is received, the task relative to this event is performed and/or the associated visualization pipeline is updated.

As described in figure 4.15, the ParaView client is composed of a GUI panel. From this panel, a user can send data and/or commands to the ParaView server, which forces the underlying VTK objects to be updated. These VTK objects trigger in turn updates to the DSM objects and service thread, which manages communication requests and data transactions with the DSM. In the following section we study common scenarios that have been implemented and are handled by the plug-in.

Workflow and Requested Notifications

In 3.1.5, we described how the architecture based on threads allows the transfer of notifications and events. To apply this architecture to the ParaView framework, we define different types of notifications:

1. `CONNECTED/DISCONNECTED`: when the simulation connects or disconnects from the DSM, this notification tells the user that the simulation has been properly connected or disconnected;
2. `WAITING`: if the simulation is paused using a *wait* command call (e.g., `H5FD_dsm_steering_wait`), this notification tells the user that the simulation is now waiting for a *resume* command;
3. `NEW_DATA`: when the DSM file is closed and new data has been written to the DSM, this notification tells the ParaView servers to update the whole post-processing and rendering pipelines;
4. `NEW_INFORMATION`: when one needs to get time information from the simulation or any other information that does not require a full pipeline update, this notification can be used to update only the information part of the pipeline.

The above notifications represent a minimal set of notifications to communicate the different simulation states or requests to the ParaView servers and GUI; there is no restriction for adding more notifications if needed.

In a common scenario (as described in figure 4.15) the simulation will connect, send a notification back to the GUI (`CONNECTED`), write data, which will in turn send another notification (`NEW_DATA`). If this is a new data (as opposed to information) notification, then the GUI schedules the update of the user instantiated pipelines (which can be a set of various source objects structured or unstructured, on top of which post-processing filters have been applied). Note that in a pull-driven system, the analysis code must query whether new data is present and if so, update its analysis pipelines. In a push-driven system, this may not be possible if some user related task is currently running on the main thread, as this will block other GUI/server requests. Using the notification thread introduced in 3.1.5, notifications are sent on a separate socket connection so that ParaView requests and push-driven DSM requests do not interfere with each other.

Then, depending on the notification that is to be received, parallel readers will be triggered and data will be read back from the DSM, allowing parallel visualization of DSM files.

4.2.2. Parallel Visualization of DSM files

One of the 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. Data written into the DSM and stored using the HDF5 API can follow any user defined patterns. A user may defined his own representation, and therefore choose a unique layout to store his data that is adapted to a given simulation. Data may thus be stored in multi-dimensional arrays (datasets) and organized in different groups so that one can easily retrieve data from.

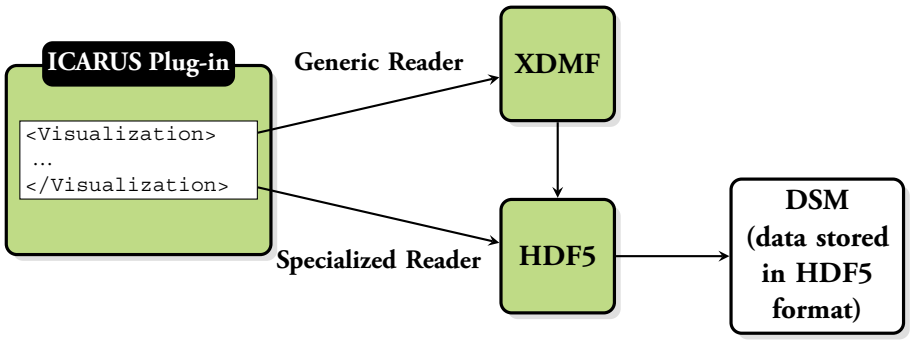


Figure 4.16. XDMF provides a generic way of reading HDF5 data stored in the DSM, while a specialized reader can provide higher performance when requested.

As a consequence (as described in figure 4.16) only two different ways of reading data can be defined, either by implementing a customized reader (which potentially means one reader per simulation and data layout) or by finding a way of dynamically describing the data (e.g., through XDMF, the eXtensible Data Model and Format).

XDMF and XDMF Readers

The eXtensible Data Model and Format (XDMF) has been designed towards that goal [21], 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 is organized in *Grids*[3], which can be seen as a container for information related to 2D and 3D points, structured or unstructured connectivity, and assigned values (e.g., pressure, velocity, etc). For each *Grid*, XDMF defines topology elements (Polyvertex, Triangle, Tetrahedron, ...),

[3]The *Grid* objects can be also described here as VTK objects [39].

which describe the general organization of the data, and geometry elements, which describe how the X, Y and Z coordinates are stored. Attributes (associated values to the mesh nodes or cells) 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 (as shown in figure 4.16, 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 [21] or use a simpler XML description file that follows the XDMF syntax, which our plug-in defines and uses, to generate a complete XDMF file on-the-fly; this point is more detailed in the following section.

XML Description Templates

To make use of the plug-in, a significant portion of the work goes into the creation of XML templates, which describe the outputs from the simulation, the parameters that may be controlled, and the inputs back to it. The XML description templates are divided in two distinct parts, one called *Visualization* describing the data for visualization only (see figure 4.16), and one called *Interaction* defining the list of steering parameters and commands one can *control* (and we will further describe it in section 4.2.3).

The XML template format that we have created does not require the size of datasets to be explicitly stated, which is particularly useful when the dataset dimensions change over time as this file is generated and read on-the-fly; only the *structure* of the data (topology/connectivity) needs to be specified along with its path to the HDF5 dataset (e.g., if a dataset named `Dataset` is stored into a group named `Group`, its pathname in the file is written as a standard POSIX pathname: `/Group/Dataset`). As the file is received, the metadata headers and self-describing nature of HDF5 datasets allow the missing information (e.g., number of elements in the arrays, precision, dimensions) to be filled-in by in-memory routines using the HDF5 `h5dump` utility (that we modified for this purpose).

The template allows one or more *Grids* to be defined, which are mapped to datasets. If the datasets written to the DSM are multi-block datasets, as many grids as the number of blocks must be defined. Each *Grid* is defined using 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.

```
<Visualization>
  <Xdmf>
    ...
    <Grid>
      <Topology TopologyType=Type>
      </Topology>
      <Geometry GeometryType=Type>
        <DataItem>HDF5 path</DataItem>
      </Geometry>
      <Attribute>
        <DataItem>HDF5 path</DataItem>
      </Attribute>
    </Grid>
    ...
  </Xdmf>
</Visualization>
```

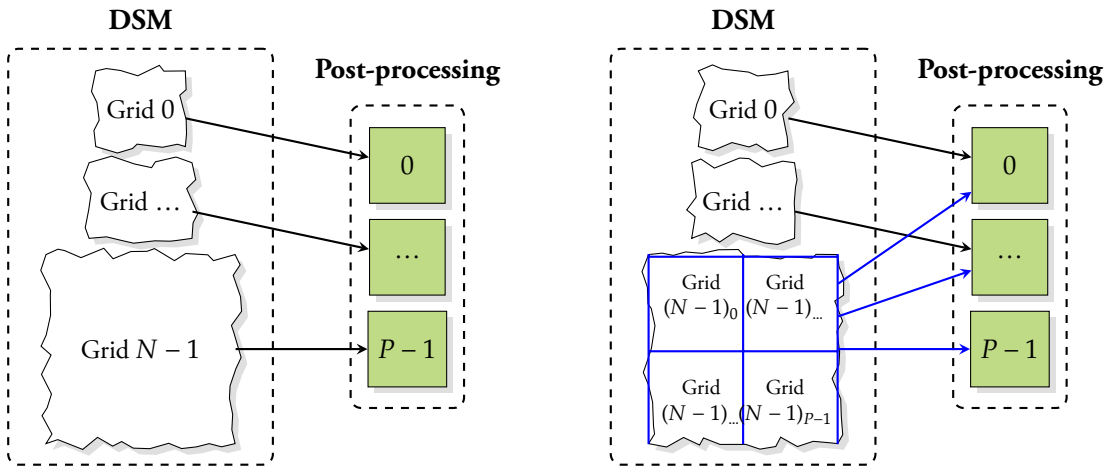
Several attributes can be added specifying for each the HDF5 path to access the data. Note that specific XDMF operations such as the `JOIN` operation can still be provided to combine individual components (e.g., velocity vector) into multi-dimensional vector arrays, for instance:

```
<Grid>
  ...
  <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>
  ...
</Grid>
```

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 (see figure 4.16). Note that only the ParaView GUI client needs access to the template as the fully generated XML is sent to the server at initialization time using the ParaView client/server communication layer (presented in section 4.2.1).

Extending Templates for Specialized Readers

Using the XDMF template description, the XDMF reader is capable of reading blocks from the DSM in parallel, making it suitable for multi-block simulations. However in some cases (as illustrated in figure 4.17a) blocks and grids are unbalanced, particularly so in SPH simulations where mesh structures and fluid particles are stored in separate grids (and the later can be of several orders of magnitude bigger than the others). Therefore to solve this potential issue, we present here an example of extension to the XDMF reader for reading particle data, serving a complementary role, directly within our plug-in.



(a) By default the XDMF reader reads data by block or grid.

(b) The H5Part reader allows redistribution of a given grid that follows the H5Part format.

Figure 4.17. Adding specialized readers for reading DSM data can effectively improve unbalanced reads of the original XDMF reader.

H5Part [3] is a library optimized for read/write of HDF5 particle data in parallel, making it suitable for reading the portion of our data using hyperslab selections[4] onto all nodes. To support two readers working on the DSM, the description of the particles is removed from the XDMF part of the XML description template and a new `<H5Part>` XML section is added, which contains the path to the particle arrays. For instance:

```
<Visualization>
  <Xdmf>
    ...
  </Xdmf>
  <H5Part Name="Particles">
```

[4] A *hyperslab* selection can be defined as a logically contiguous collection of points, or as a regular pattern of points or blocks in a memory space.

```
    <Step Name="Step"/>
    <Xarray Name="X"/>
    <Yarray Name="Y"/>
    <Zarray Name="Z"/>
  </H5Part>
</Visualization>
```

Providing both readers open the file in read-only mode, there is no danger of data corruption and the management of file and dataset handles can be left to the HDF5 layer. There is actually no restriction on how many tools or utility libraries we may use providing they adhere to the HDF5 API. As illustrated in figure 4.17b, the grid that contains particle data is subdivided into P blocks, where P is the number of processes that are used for reading.

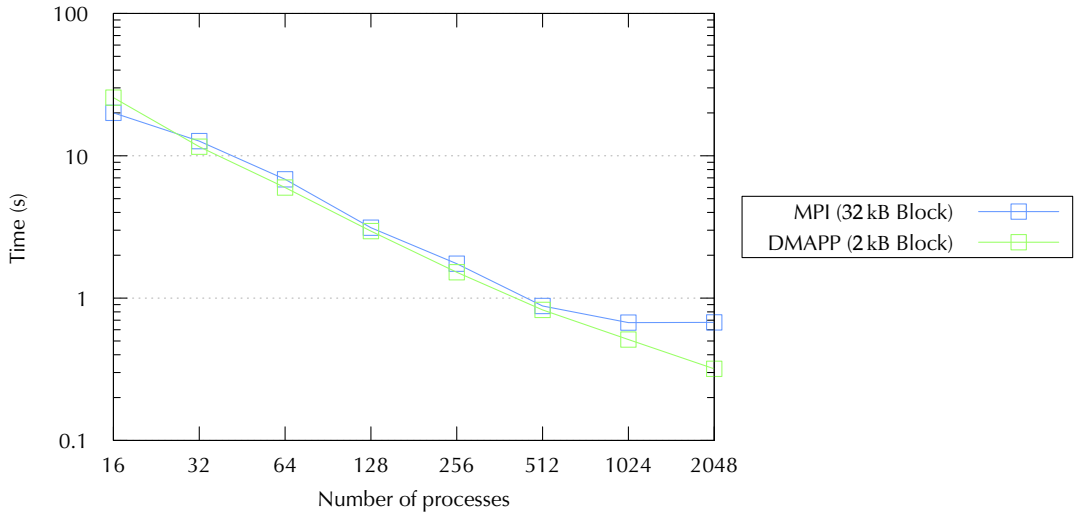


Figure 4.18. Read time on Cray XK6 using the H5Part reader of 4.92×10^8 particles (40 GB) using block-cyclic redistribution with MPI and DMAPP inter-communicators.

As we read in parallel and as data is better balanced between processes, the performance of the H5Part library on data in DSM is very good. Figure 4.18 shows the read time for a dataset of 4.92×10^8 particles on different process counts using the MPI RMA and DMAPP communicators on Cray XK6. The dataset size is 40 GB and can be read from memory in sub-second time for a number of processes greater than 512. Not using this specialized reader and therefore the XDMF reader would in this case mean make a serial read of about 40 GB, which would not even fit into the memory of the node receiving the data.

When working *live* with very large datasets, the read time is a major factor in influencing the interactive experience and thus being able to read efficiently in parallel is a factor that one should not consider as negligible.

4.2.3. Parallel Steering and Analysis

For parallel in-situ visualization, we defined in the previous section an XML description template that allows generic coupling of any simulation code to the ICARUS plug-in. For steering, we defined in 3.3.2 an interface (which is implemented on top of HDF5 and of the `dsm` driver) that one can use in a simulation code to exchange commands and data with a post-processing application. In this section we present how a user can modify simulation parameters and data generically from the plug-in, without any recompilation of the ParaView graphical user interface.

Extending Templates for Steering

To define steering parameters than can be easily set from the ParaView GUI, we follow the existing model of the ParaView server manager properties (see figure 4.14) to enable automatic generation of controls on top of the existing mechanism, which are originally used by ParaView to generate filter and source panels. We therefore add a new XML section into the original description template called `<Interaction>` section, so that we define an XML file mainly composed of two different sections, one for visualization (as we saw in 4.2.2) and one for steering commands:

```
<Icarus>
  <Visualization>
    ...
  </Visualization>
  <Interaction>
    ...
  </Interaction>
</Icarus>
```

Within this `<Interaction>` section, we allow the following list of *properties* to be defined:

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

`Int/Double/StringVectorProperties` allow scalar, vector and string parameters to be defined and generated in the GUI and are exactly the same as the existing ParaView properties (presented in 4.2.1 and that are used to transfer data between the GUI and the server). 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, constraint domains can be attached to these properties; this allows a user to restrict the parameters defined to either a boolean domain,

which will then generate a check box, or to a range domain using a $[min, max]$ interval, which will appear as a slider (see figure 4.19).

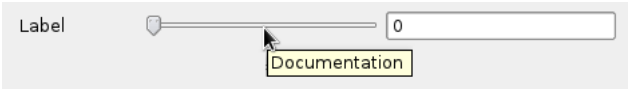


Figure 4.19. Generated control example using a range domain of $[0, x]$, label and documentation can be customized by the user.

So far two extra *properties* have been added to the ParaView server manager properties to support additional steering capabilities. 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 *Interaction* group (see 3.3.3) of the DSM file and can be checked by the simulation. It can be used to tell the simulation to perform some user defined action (e.g., a reload mesh command) and can be defined as follows:

```
<CommandProperty
  name="ReloadMesh"
  label="Reload mesh">
</CommandProperty>
```

The other, `DataExportProperty`, defines an input back to the simulation. It allows a whole ParaView dataset or a single data array to be exported into the in-memory DSM file. One may interactively select a pipeline object (i.e., a filter or a source object such as a sphere, cube, etc, or a more complex object), select the corresponding array (points, connectivity or node/cell field) and write it back to the DSM. The corresponding HDF5 path must be specified so that the location of the written array is consistent with the simulation expectations. If the array is going to be a modified version of the 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 4.20. 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 DSM library. The simulation gets the parameters and the commands by reading them from the DSM using the same names as specified in the template. Note that in this case, a `DataExportProperty` is generally bound to a `CommandProperty`, so that data is written and a command to perform some action with it is sent at the same time.

If a grid exported by the simulation is to be modified directly and then returned back to the simulation, some action to be performed may be specified in the template and reference the grid in question (e.g., if one wishes to modify the geometry of a body and we therefore bind a 3D interactive transform widget to it as shown in figure 4.21). This is done by adding hints to the properties (as below). Currently, any 3D widget may be created (e.g., box, plane, point, etc) and

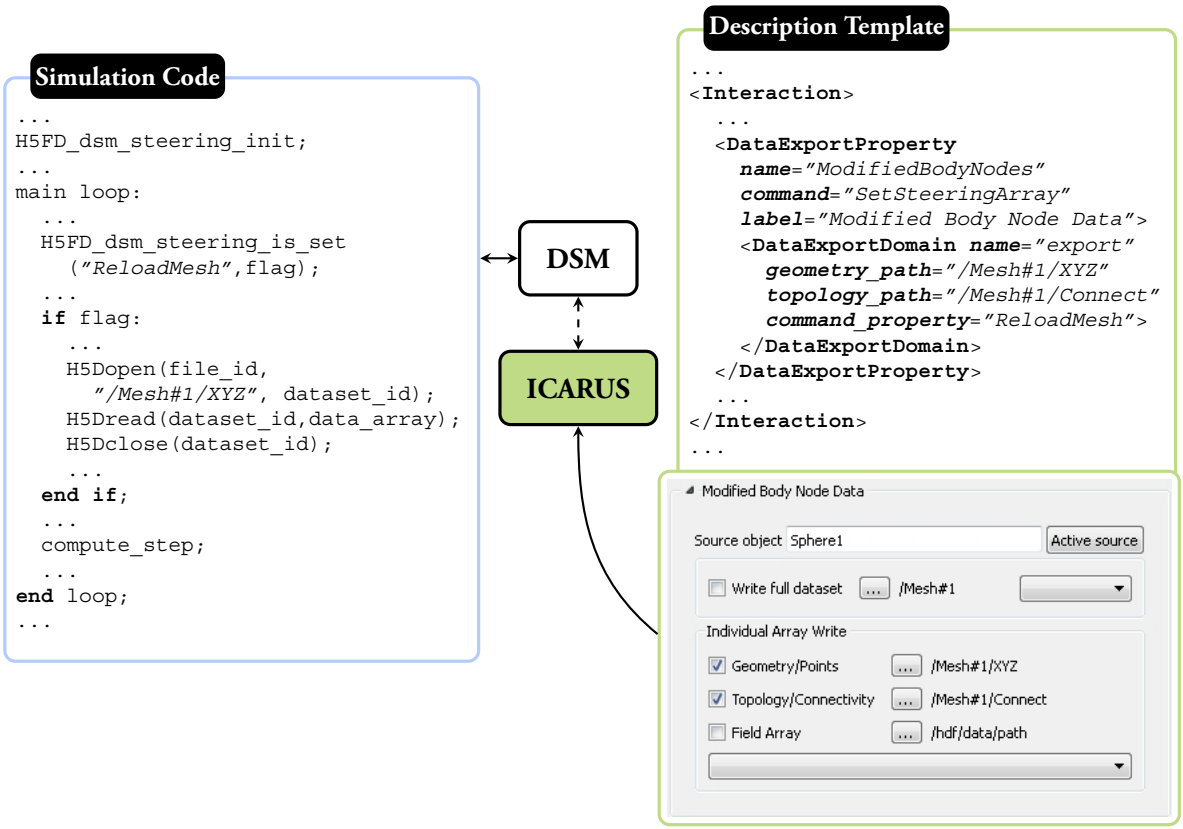


Figure 4.20. Steering usage example between a simulation code and ParaView.

for each grid with an attached widget, a mini-pipeline is instantiated containing a set of filters, which extract the dataset from the multiblock input (if multiple grids exist) and bind the widget with an associated transform to it.

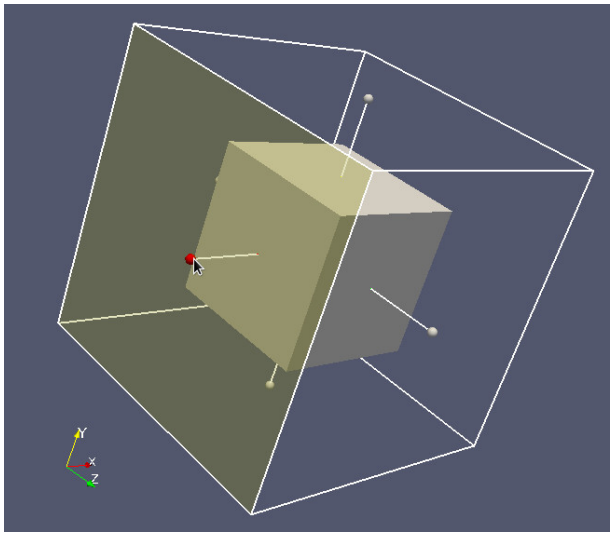


Figure 4.21. 3D transform widget example bound to a box source.

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>
```

Note that the XML templates are loaded at run time and ParaView client/server wrappers for control *properties* 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.

Parallel Steering and Overheads

Once one can generate controls from the GUI, it is necessary to add the corresponding and appropriate calls in the simulation to receive the commands and data that have been sent back. As

shown in figure 4.20, when a parameter is set in the GUI, one has to check from the simulation if this same parameter is set and if it is set, read it. In a typical and minimal steered simulation loop, one would therefore check if commands are set, compute and write data output (if needed). Therefore the overhead added to a computation step without any specific interaction is the amount of time spent to check the presence of commands, which requires a DSM file open, followed by a read and a close (which are all HDF5 operations that make use of the `dsm` driver presented in section 4.1). Note also that checking the presence of one single parameter in parallel may potentially involve thousands of processes reading data from the DSM distributed among a few hundreds of processes. As reading data from the in-memory file does not modify metadata, it is allowed in our implementation to have multiple processes doing independent reads. However this case may potentially result in thousands of requests sent to the DSM at the same time, which may create a bottleneck. To prevent this potential issue, one may thus access the file from one process and then broadcast the retrieved value to the other processes that need this same piece of information.

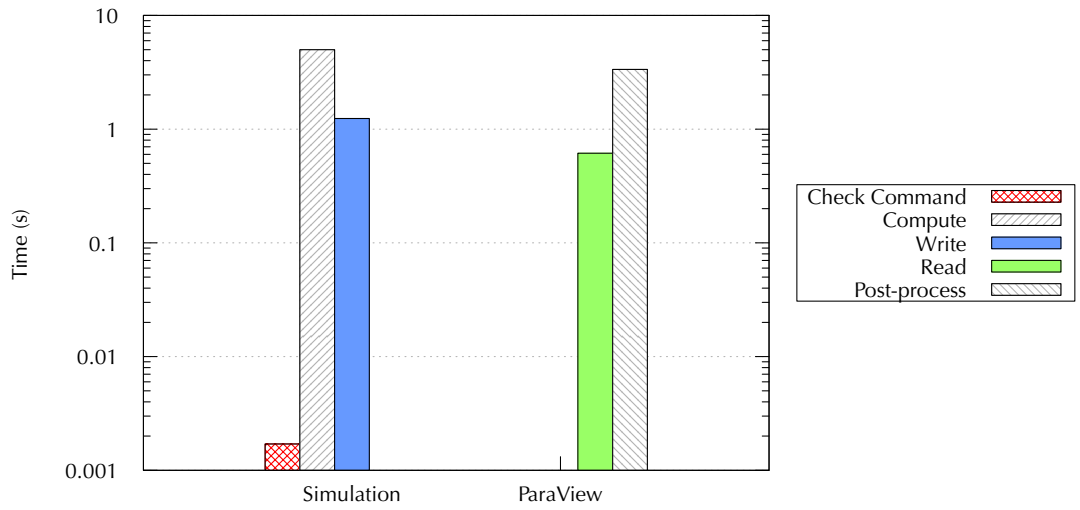


Figure 4.22. Amount of time spent in the various components of the interface by a simulation writing 40 GB files to DSM using block-cylic redistribution and DMAPP with ParaView on Cray XK6.

By making use of the same large dataset (40 GB) that we used in the previous sections (see section 4.2.2), and which contains 4.92×10^8 particles, we show in figure 4.22 the amount of time spent in the different parts of the pipeline using the DMAPP inter-communicator presented in the previous sections (as this communicator allows the highest transfer rate and lowest latency on Cray XK6). Again, to be able to steer the simulation, we must add in the simulation code calls (such as `h5fd_dsm_steering_scalar/vector_get`) to check for commands and parameters that have potentially been sent back to the DSM from the ICARUS plug-in. This time corresponds to the *CheckCommand* plot and its order of magnitude is the millisecond. *Write* operations from the

simulation to the DSM is on the order of the second or sub-second. As we described in 4.2.1, new data received from the DSM triggers a `NEW_DATA` notification, which updates asynchronously the ParaView pipeline (this implies a parallel read followed by post-processing operations). *Read* operations from the DSM to ParaView respects the same order of magnitude and is generally faster (see 4.1.4) than writing to the DSM, as DSM hosts and ParaView servers share the same nodes. In this particular example the bottleneck of this loop is the *Compute* time which is here on the order of 1×10^1s (the simulation that has been used for this test is presented in more details in 5.1). Writing and reading to the DSM is an asynchronous operation and this means that simulation and ParaView servers operate asynchronously on the DSM data. Therefore the total time spent in updating the visualization pipeline depends here on the time spent by the simulation to compute a time step.

4.2.4. Integration Conclusions

In this chapter, we presented a plug-in called ICARUS, which enables integration of our framework into the well-known ParaView application. By making use of XML templates, which allow description of a very large variety of data models and layouts, we showed how one can simply interface a simulation code that writes data using the HDF5 library to our plug-in, removing also the need from specifying data dimension and precision. For efficiency and better load balancing of read operations, we also showed how one can extend XML description templates to make use of specialized readers, such as the H5Part reader that allows gigabytes of particle data to be read in sub-second time. The ability to make use of multiple custom readers which have been tuned for a particular kind of data layout within the same framework demonstrates the flexibility of the approach and the ease with which it may be extended. Any reader that is implemented on top of the HDF5 library can be easily modified to make use of the `dsm` driver (as we did for XDMF, H5Part) and allows the combination of the usual ParaView filters and post-processing tools in the ParaView pipeline without any particular restriction. By fine tuning data transfers using Cray DMAPP libraries as well as the simulation and analysis I/Os, we can therefore minimize the delay experienced by the user before performing analysis tasks on their data.

For steering, we showed how one can generate different types of controls by extending the description templates with an *Interaction* section. Controls are generated on-the-fly and do not require any recompilation or re-linking of the libraries. The framework is therefore generic enough and can be used *as is* with any simulation code. Sending commands back to the simulation requires a minimal overhead, which depends on the communication system used and can be negligible, especially for simulations where the computation time of one time step exceeds the order of the second. More complex controls such as the re-meshing of objects can increase the transfer time

depending on the resolution of the object that is sent back. In the next chapter we present concrete usage examples through two different simulation codes.

Chapter 5.

Application on SPH Simulations: Model Validation

Utilization of a common framework for in-situ visualization within a broad range of simulation applications was one of the main objectives for this work. While a large effort for adapting and modifying the structure of the simulation codes can be required for steering and enabling modification of objects (e.g., a re-meshing operation), we showed in the previous chapter that re-routing data to the DSM for in-transit visualization purposes only requires one line of code to be modified (for simulation codes that already make use of HDF5). In this chapter, we show how one can make use of the platform presented in chapter 4 to: visualize and analyze *in-situ* data coming from the simulation while removing the disk I/O bottleneck; easily send steering commands as well as steering data for object modification back to the simulation. We choose in this chapter two simulation codes from the NextMuSE [24] European project and study in details in 5.1 and in 5.2 the integration of our framework, as well as the advantages brought by our approach. We finally illustrate in 5.3 the validation of our model through some of the NextMuSE project milestones that were to be reached.

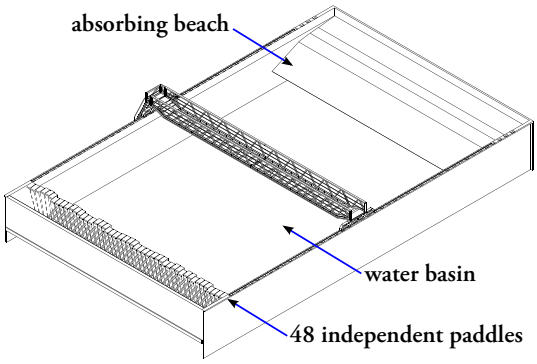
Introduction to NextMuSE Test Cases

As explained in 1.2, the NextMuSE [24] European project defines a multidisciplinary environment between different simulation codes, which may use different data representations, in various domains such as energy, transport and health-care. In the following sections, we demonstrate the integration of our framework into two different simulations codes: one capable of running on a large number of CPUs, producing large amounts of data but at a low frequency; one running on GPU, producing smaller amounts of data but at a higher frequency. Several computational fluid dynamic models and particularly SPH models now make use of GPGPU computing (General-Purpose Computation on Graphics Hardware [35]). This is for example the case of [31] where a

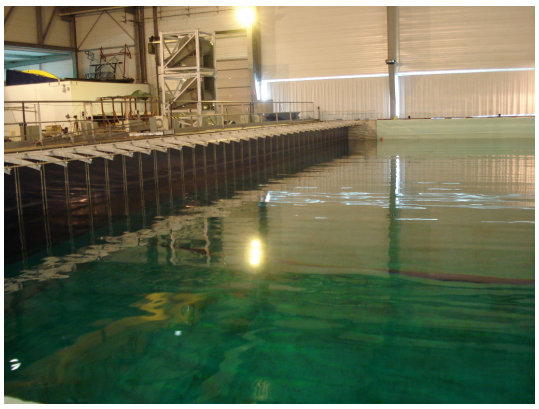
very interactive simulation can be obtained and rendered using shaders or ray-tracing creating the effect of a real fluid. To obtain such a level of interactivity, the number of particles is generally decreased or the precision of the results and SPH computation models are less accurate to decrease the number of computational operations. In the following test cases, as per the NextMuSE objectives (see 5.3) that are defined for high performance computing, we consider a relatively high number of particles (i.e., in the order of 10 million particles, even if as we will see, the GPU simulation test case does not reach this order of magnitude yet). We can therefore keep a reasonably good level of interactivity (on the current hardware) while having a good degree of accuracy in our results.

5.1. Integrating ICARUS into SPH-flow

The solver we use here is designed for CPU computing and uses several different models that provide a high degree of accuracy, which of course have the consequence that the more precision requested, the lower the interactivity. This solver, called SPH-flow [58] and developed by ECN (Ecole Centrale de Nantes) and HydrOcean, is able to compute fluid and multi-physic simulations involving structures, fluid-structure couplings, 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 mesh-based solvers cannot handle easily. A significant effort has been made in the context of NextMuSE to improve the SPH model towards more accuracy and robustness, together with high performance on thousands of processors.



(a) ECN wave tank, $50 \times 30 \times 5$ m.



(b) 48 independent flap wave-maker.

Figure 5.1. SPH-flow test case.

The test case that we use for this code is a wave tank simulation, based on a real model that was built at ECN. The ECN wave tank [27] is a water basin composed of a 48 independent paddle wave-maker and of an absorbing beach (see figure 5.1). Several types of waves can be generated, their amplitude and frequency varying by modifying the position and speed of the paddles to simulate real sea conditions. Objects such as ship/hull models are usually placed into the tank so that experiments and measures can be conducted to determine the behavior of the ship under certain wave patterns.

Using the SPH-flow solver, the same type of experiment can be modeled and one can therefore compare the real and simulated test cases. We focus in the following section on this experiment: in 5.1.1, on the in-situ visualization aspects and in 5.1.2, on the steering and re-meshing aspects. We conclude on the integration of our framework in 5.1.3 and present on-going developments. However as of the date this thesis is written, the simulation code cannot handle multiple paddles and instead of a wave maker composed of 48 paddles, we consider in the following sections a simplified test case that makes use of only one single paddle.

5.1.1. In-situ Visualization

As previously said the test case used here is a wave maker. Three different types of *objects* are therefore defined in the simulation: the particles representing the fluid, the tank (mainly the application domain boundaries), the paddle of the wave-maker, an object (cube, sphere, etc) falling into the fluid. For in-situ visualization, as the simulation writes data into HDF5 files, one may therefore consider the hierarchical structure of figure 5.2.

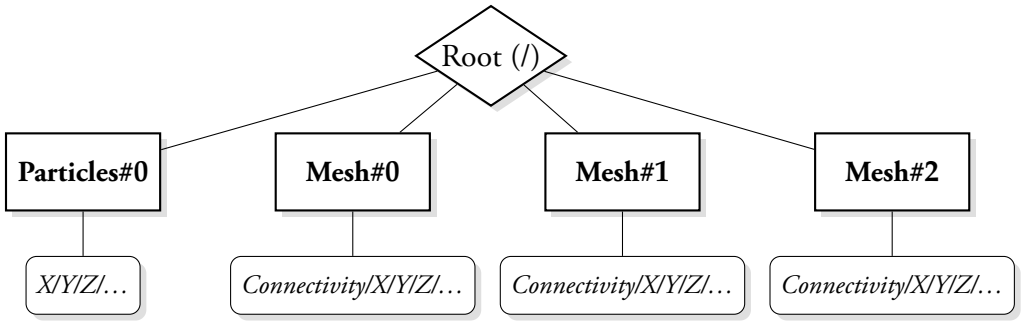


Figure 5.2. Hierarchical approach representation of SPH-flow data (for clarity, other attribute arrays such as pressure, velocity, etc, are not represented but are stored at the same level as the point and connectivity arrays).

The fluid particles are represented as X, Y, Z coordinates and are stored using the H5Part [3] description in a group called Particles#0 (so that we can make use of the H5Part reader approach presented in 4.2.2 to read particle data more efficiently in parallel from the DSM). Other objects

are stored using regular meshes in separate groups, respectively Mesh#0, Mesh#1 and Mesh#2 so that we can easily access and modify them if necessary as we will see in 5.1.2. Once this structure defined, data can be written to the DSM by switching from the regular HDF5 `mpio` driver to the `dsm` driver. As writing to the DSM is an operation that does not involve participation of the post-processing side for synchronization, the simulation can write data and exit from the I/O routines without waiting for the post-processing operations to complete.

An overview of the code (Fortran) is given in annex D.1. Since the code was already able to write data using the HDF5 format, switching to the `dsm` driver was a quick and easy operation as it requires only the addition of an `h5pset_fapl_dsm_f` call. Note that the SPH-flow simulation code can also compute and write inside the main I/O routine isosurfaces (using a marching cube [45] technique) so that one can easily visualize the surface of the fluid. This extra data was originally written to a separate file. However, because the `dsm` driver can only store a single file at the moment, this unfortunately caused problems. By default, a file created in the DSM triggers a wipe of the memory (as opposed to a file open). Consequently, the second file written (corresponding to the marching cube output) 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 no in-situ visualization capability is required). This represented the only change we add to make to enable in-situ visualization of SPH-flow data. In future versions of the driver, adding support for multiple file storage would make this operation even easier.

Writing to the DSM

To determine the performance one can get when writing to the DSM using the HDF5 `dsm` driver with SPH-flow, the simulation is executed on the same Cray XK6 machine used in the previous chapter and with the same configuration. To study the scalability (as a reminder), the number of processes is increased, keeping a constant number of processes per node until the number of nodes available is exhausted. Therefore the number of processes on the XK6 used in this configuration for writing is 4 per node until 128 nodes are reached (512 processes), at which point to saturate the bandwidth the number of processes per simulation node is increased up to 16 (giving 2048 processes writing data in total). The DSM (which receives data) is distributed among 40 nodes (160 processes, 4 per node). The resulting write time is represented in figure 5.3. Note that the write time represents the time spent in the I/O routines of the SPH-flow code and cannot be directly associated to a write bandwidth result (as extra allocations, memory copies, etc, are performed). In this particular example 20×10^6 particles are written, which gives a total file size of about 1.8 GB.

One can easily notice that writing to the DSM is always a cheaper operation than writing to the file system, and more importantly than computing a time step. Also while we reach the

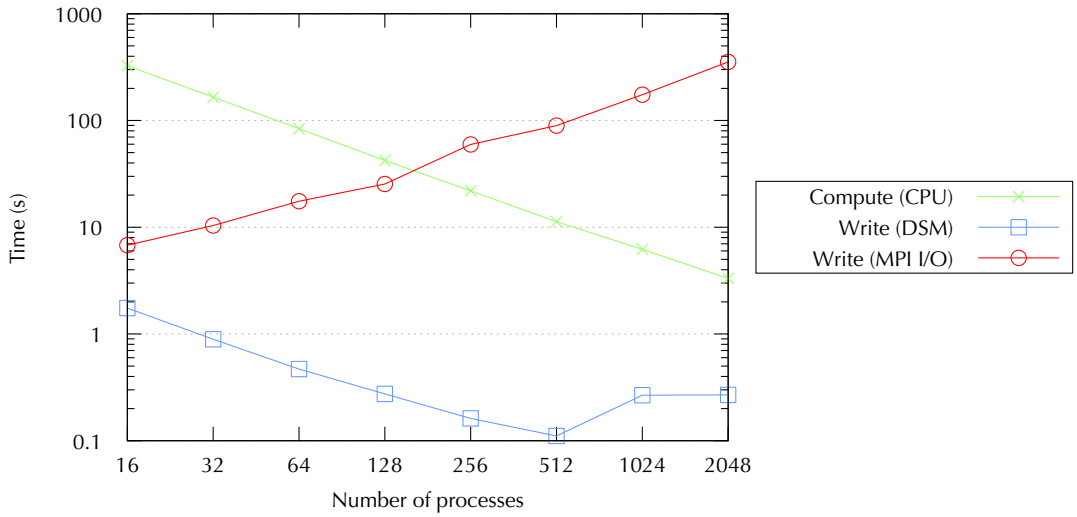


Figure 5.3. Compute and write time on Cray XK6 of 20×10^6 particles using SPH-flow—Comparison between I/Os to disk and DSM (block-cyclic redistribution with DMAPP inter-communicator).

maximum number of nodes for 512 processes, we keep a good scalability up to this point. When the number of processes per node is increased to saturate the bandwidth, one can see that the performance decreases, but without showing any significant drop point. Also notice that writing to the file system performs very poorly in this case (which is mainly due to the small number of Lustre object storage targets available and to configuration/hardware issues). Even in a better configuration such as the one presented in [81], writing to the file system can still represent a real bottleneck, particularly so when the file system is saturated, whereas there is no real hardware limitation (except the total number of nodes available) for the number of DSM nodes used for receiving data from the simulation. The bottleneck becomes even more visible when the frequency of outputs is high, as the time spent for writing may become larger than the computation between two writing operations. Being able to redirect data to a distributed shared memory allows us in this case to remove a significant bottleneck, as well as giving us the ability to visualize and analyze data in real-time.

Reading from ICARUS

Once data is written to the DSM, the hierarchical description stored in the HDF5 file metadata does not allow the reader to distinguish position arrays from velocity arrays (unless specific names or groups are used, which would require usage of a particular layout, such as the one introduced by H5Part [3]). To enable the use of the XDMF reader presented in 4.2.2, the following XML description file (which describes the topology, geometry and attributes defined in the HDF5 file)

must therefore be passed to the ICARUS ParaView plug-in (only the visualization part of the XML description is presented here, the interaction part is presented in 5.1.2).

```
<Icarus>
  <Visualization>
    <Xdmf>
      <Grid Name="Particles">
        <Topology TopologyType="Polyvertex"/>
        <Geometry GeometryType="X_Y_Z">
          <DataItem Name="X">/Particles#0/X</DataItem>
          <DataItem Name="Y">/Particles#0/Y</DataItem>
          <DataItem Name="Z">/Particles#0/Z</DataItem>
        </Geometry>
      </Grid>
      <Grid Name="Tank">
        <Topology TopologyType="Triangle" BaseOffset="1">
          <DataItem Name="Connectivity">/Mesh#0/Connectivity</DataItem>
        </Topology>
        <Geometry GeometryType="X_Y_Z">
          <DataItem Name="X">/Mesh#0/X</DataItem>
          <DataItem Name="Y">/Mesh#0/Y</DataItem>
          <DataItem Name="Z">/Mesh#0/Z</DataItem>
        </Geometry>
      </Grid>
      <Grid Name="Body">
        <Topology TopologyType="Triangle" BaseOffset="1">
          <DataItem Name="Connectivity">/Mesh#1/Connectivity</DataItem>
        </Topology>
        <Geometry GeometryType="X_Y_Z">
          <DataItem Name="X">/Mesh#1/X</DataItem>
          <DataItem Name="Y">/Mesh#1/Y</DataItem>
          <DataItem Name="Z">/Mesh#1/Z</DataItem>
        </Geometry>
      </Grid>
      <Grid Name="Wave Maker">
        <Topology TopologyType="Triangle" BaseOffset="1">
          <DataItem Name="Connectivity">/Mesh#2/Connectivity</DataItem>
        </Topology>
        <Geometry GeometryType="X_Y_Z">
          <DataItem Name="X">/Mesh#2/X</DataItem>
          <DataItem Name="Y">/Mesh#2/Y</DataItem>
          <DataItem Name="Z">/Mesh#2/Z</DataItem>
        </Geometry>
      </Grid>
    </Xdmf>
  </Visualization>
```

```
</Icarus>
```

This description can actually be seen as a simple way of associating the hierarchical structure of figure 5.2 to the XDMF reader. Four *Grids* are defined, one per object type; this allows the definition and usage of different data representations, which can be structured or unstructured. Data stored in the DSM does not have to follow a unique data model and one can store particle data as well as mesh data with different storage layout conventions. As explained in 4.2.2, the XDMF reader reads data in parallel by grids (i.e., XDMF grids are distributed among processes, which read pieces of data corresponding to that particular grid). In this case, considering N processes are used for parallel visualization, only the first four processes will be used for reading data. By using the H5Part reader for the particle grid, data from this grid can be read in parallel using each of the N processes. The following XML description is therefore preferably used instead:

```
<Icarus>
  <Visualization>
    <H5Part Name="Particles">
      <Step Name="Particles"/>
      <Xarray Name="X"/>
      <Yarray Name="Y"/>
      <Zarray Name="Z"/>
    </H5Part>
    <Xdmf>
      <Grid Name="Tank">
        <Topology TopologyType="Triangle" BaseOffset="1">
          <DataItem Name="Connectivity">/Mesh#0/Connectivity</DataItem>
        </Topology>
        <Geometry GeometryType="X_Y_Z">
          <DataItem Name="X">/Mesh#0/X</DataItem>
          <DataItem Name="Y">/Mesh#0/Y</DataItem>
          <DataItem Name="Z">/Mesh#0/Z</DataItem>
        </Geometry>
      </Grid>
      ...
    </Xdmf>
  </Visualization>
</Icarus>
```

Once this description passed to the ICARUS plug-in, a user can interact with the data, add analysis filters; the instantiated ParaView pipeline is automatically updated as new data is received. Figure 5.4 shows an example where a cube is falling into the tank while the wave maker is starting creating waves. As previously said, one can see in this figure that both mesh and particle data are used, allowing the model to use the most adapted types of representation.

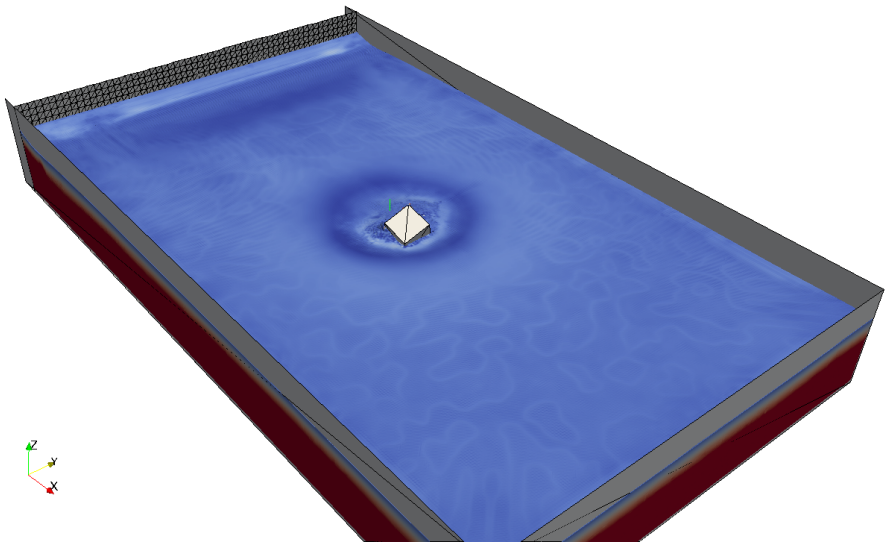


Figure 5.4. In-situ visualization of SPH-flow data (and marching cube output) using 20×10^6 particles on Cray XK6. Pressure values are represented here.

5.1.2. Computational Steering

Once one can visualize data in-situ, the next step of integration is the insertion of steering capabilities into the simulation code. In the context of SPH-flow several parameters can be modified. It may for example be interesting to make dynamically vary the frequency and the amplitude of the wave maker, or experiment the effect of the generated waves onto an object placed into the tank without having to relaunch the simulation. It may also be interesting for one to modify the original object during computation, thereby altering its shape or increasing its resolution on demand.

Defining Steering Commands

To define the commands that will be available for a user, the XML template description file (which needs to be passed to the ICARUS plug-in) must be extended with an `<Interaction>` section (see 4.2.3). In this example, we consider the case where an object placed into the tank can be dynamically re-meshed and the force applied to this object as well as its momentum can be interactively modified (so that the object can be moved in the tank). To modify the force and the momentum, simple vectors that contain the X, Y and Z components can be sent back, which can be defined by adding `<DoubleVectorProperty>` tags into the XML description file. Again, choosing an appropriate name is important, as this name will be used by the simulation to get data back from the DSM (therefore this name must be unique). As these vectors are three component vectors, the number of elements is set to 3. Default values are set to zero. Note that no range

domain is specified here, so any double value can be sent back to the DSM. The XML sections are defined below:

```
<Icarus>
  <Visualization>
    ...
  </Visualization>
  <Interaction>
    ...
    <DoubleVectorProperty
      name="Force"
      command="SetSteeringValueDouble"
      label="Force on the free body"
      number_of_elements="3"
      default_values="0.0 0.0 0.0">
      <Documentation>
        Set the force of the free body
      </Documentation>
    </DoubleVectorProperty>
    <DoubleVectorProperty
      name="Momentum"
      command="SetSteeringValueDouble"
      label="Momentum on the free body"
      number_of_elements="3"
      default_values="0.0 0.0 0.0">
      <Documentation>
        Set the momentum of the free body
      </Documentation>
    </DoubleVectorProperty>
    ...
  </Interaction>
</Icarus>
```

To modify the object (which is stored in a group called Mesh#1 in figure 5.2), two additional sections are added. The first one allows the detection of a user command while the second one defines the HDF5 paths to the new geometry arrays that are to be written into the DSM. In this case, the same group Mesh#1 is re-used and new data is written into arrays called `NewXYZ` and `NewConnectivity` (which store geometry and connectivity data respectively). The original data arrays are not overwritten, for implementation reasons it is easier here to simply add new arrays (that have different dimensions) rather than having to delete the objects from the file and recreate them (but nothing prevents us from doing it). The XML sections are defined below:

```
<Icarus>
  <Visualization>
    ...
```

```
  </Visualization>
  <Interaction>
    ...
    <CommandProperty
      name="ReloadMesh"
      label="Reload mesh"
      command="ExecuteSteeringCommand"
      si_class="vtkSIProperty">
    </CommandProperty>

    <DataExportProperty
      name="ModifiedBodyNodes"
      command="SetSteeringArray"
      label="Modified Body Node Data">
      <DataExportDomain name="data_export"
        geometry_path="/Mesh#1/NewXYZ"
        topology_path="/Mesh#1/NewConnectivity"
        command_property="ReloadMesh">
      </DataExportDomain>
    </DataExportProperty>
    ...
  </Interaction>
</Icarus>
```

Modifying SPH-flow

Once this set of commands is defined in ICARUS, SPH-flow must be modified accordingly. Adding simple computational steering to the simulation did not require significant effort (see annex D.1). On start, calls to set the time range of the simulation were inserted, which actually set the start and end time values of the simulation in ParaView. Passing this information to ParaView is very important for steering as it allows us to create time dependent interactions using *animation keyframes*.

The ParaView animation keyframe editor is a very useful tool that can be used for steering objects and parameters of the simulation. As illustrated in figure 5.5, setting the time range in the simulation updates the start and end time values in the animation panel. Then parameters can be animated depending on the current time value that is picked up from the simulation. If we consider a simple case where a parameter varies from a value a to a value b between t_1 and t_2, a keyframe can be defined, which then automatically interpolates the value of the parameter at the current time step (using a given interpolation function). More complex scenarios can be defined and transform filters can be applied to for example modify the position or the shape of an object

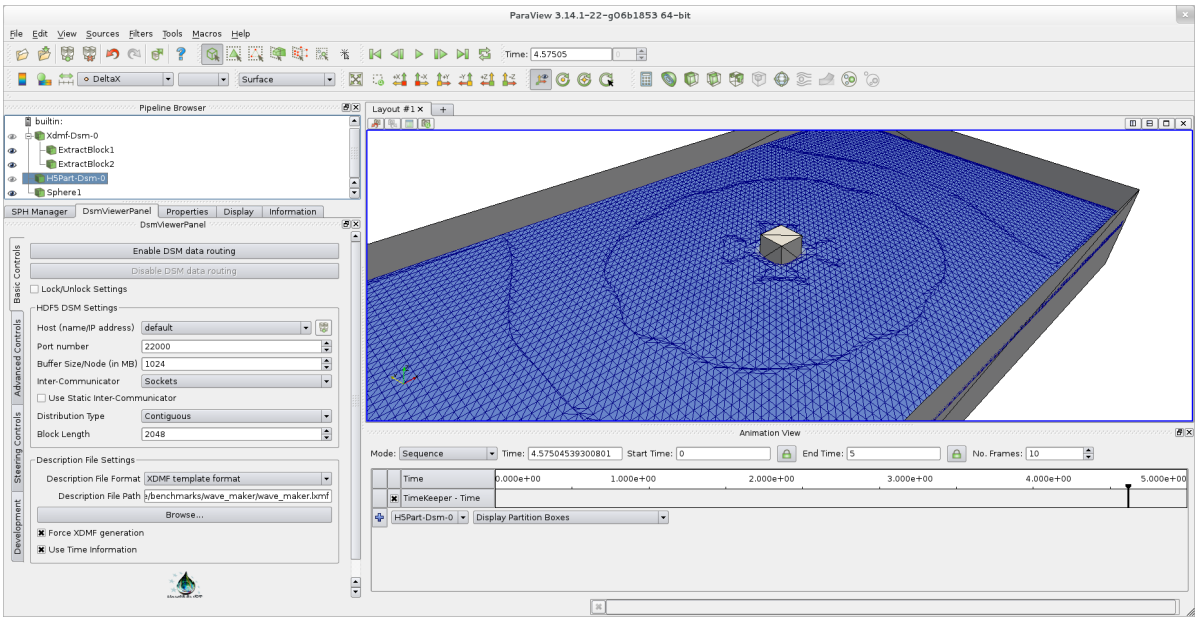


Figure 5.5. Time range is set in the animation view at the initialization of SPH-flow, which allows us to make use of animation keyframes for steering objects. As the current time value is incremented, the animation view is updated.

over time. We can therefore create semi-automatic interactions, that allow automatic reexport of values into the simulation. In the simulation code, modifying parameters such as the wave maker frequency or object momentum becomes trivial in comparison as it requires only simple `h5fd_dsm_steering_scalar/vector_get` calls with the name of the arrays defined in the XML description.

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; it represents an entirely new development in the simulation code and care must be taken that when the new geometry appears, the associated variables do not remain uninitialized, which would cause the simulation to blow up. The strategy of figure 5.6 is used.

Once the simulation is initialized and has started the computation loop, we add a call to `h5fd_dsm_steering_is_set` to check the presence of a *ReloadMesh* command. If this command is set, data corresponding to the object that is to be reloaded is read from the DSM, and the object kinematics are redefined (gravity center position, velocities, etc). Faces and normals to the object are then re-initialized. If the command is not set, the simulation carries on computation until it reaches convergence (which is in this case the simulation physical end time value defined by the user) or until a new *ReloadMesh* command is found.

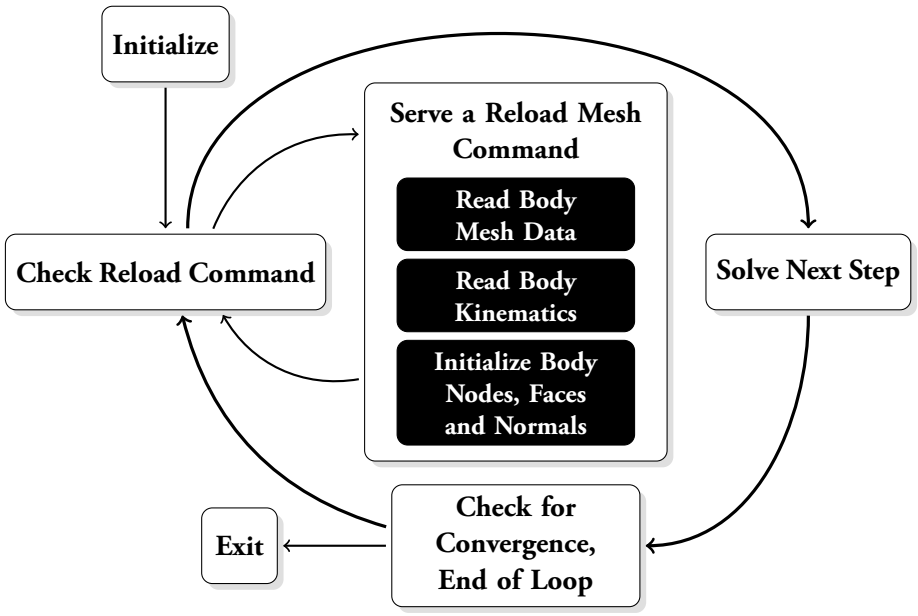


Figure 5.6. SPH-flow computing loop with mesh reload capability.

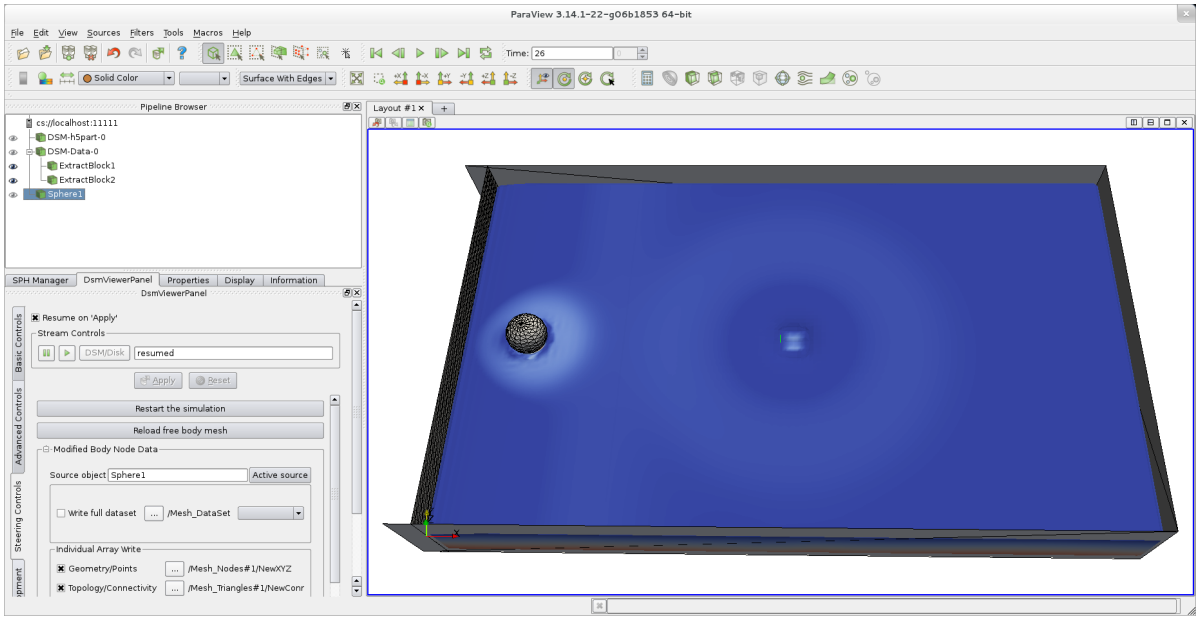
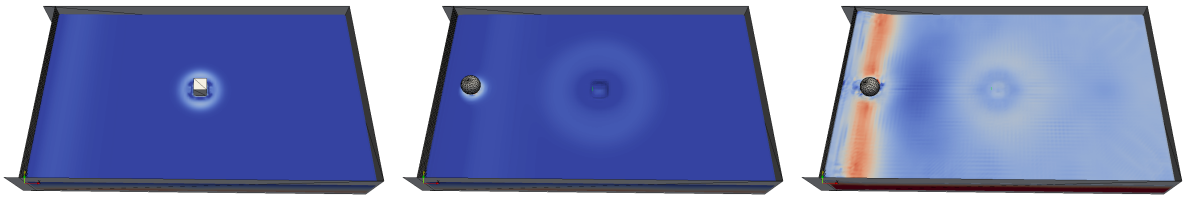


Figure 5.7. The interface generated for SPH-flow using the previously described XML template. The bottom left panel contains the generated GUI that is used to enter/modify parameters to reload the sphere into the simulation. The sphere is written into the DSM in parallel.



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

Figure 5.8. Three images from a sequence where a sphere is re-injected by ParaView and written into the DSM where it is reread by the SPH-flow simulation.

Figure 5.7 shows a screenshot of an SPH-flow steering session. We are able to pause/resume the simulation (not necessary but more convenient) and place an arbitrary geometry in instead of the cube, using a reload command. For illustration, we show a sphere generated on 4 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 dataset. The sequence of images in figure 5.8 shows snapshots from an accompanying animation. At t_0 the cube is falling into the fluid. At t_1 the sphere has been re-injected into the simulation and starts interacting with the fluid. Notice that the impression of the cube object that was present in the fluid when the simulation was initialized is still present and is progressively disappearing. At t_2 the wave created by the wave maker is interacting with the newly introduced object (i.e., the sphere). As the simulation follows the model of figure 5.6, nothing prevents us from reloading another object during the same run or reload the same object from ParaView using animation keyframes (see [81]), the object being moved or deformed according to predefined paths, which can be adjusted on the fly.

5.1.3. Conclusion and Future Developments

In the previous section, we showed how one can make use of the ICARUS framework to reload a mesh on the fly and to dynamically replace an object moving in the tank (a real example would be to replace the cube with a hull). 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 the features inside the simulation and it makes more sense to couple an application (such as ParaView) dedicated to these tasks to it. To this end, we also sought to implement a wave maker which could be controlled by the user to produce a variety of conditions under which the models can be tested.

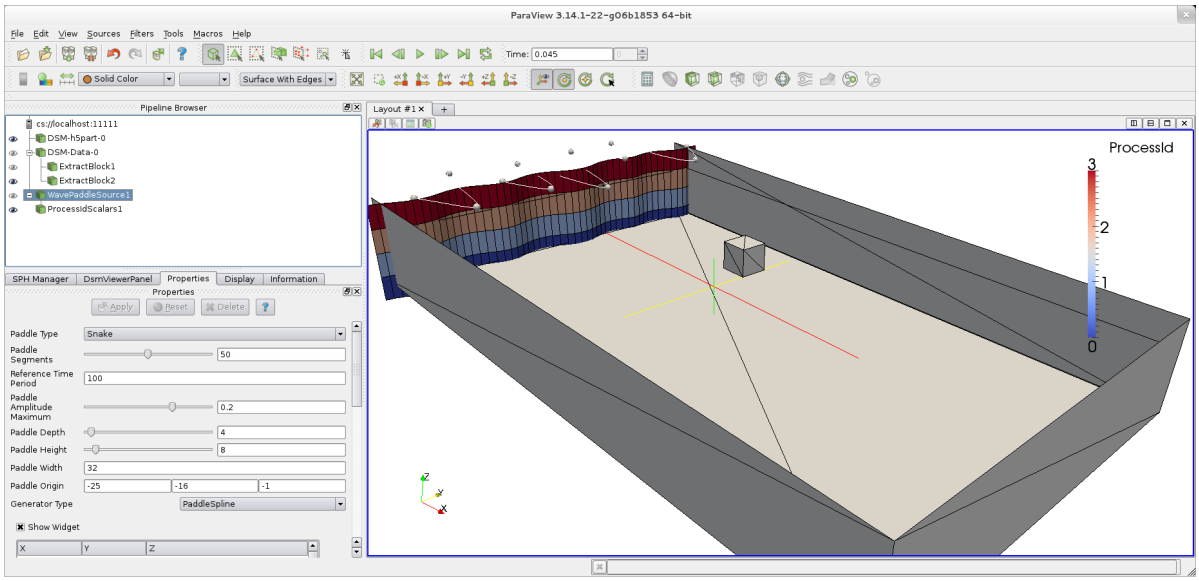
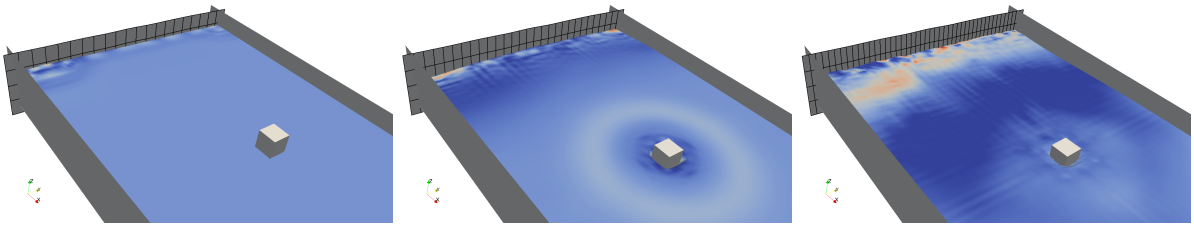


Figure 5.9. A custom control for shaping a wave maker in ParaView. The paddle is generated and distributed on 4 processes and re-loaded into the simulation in parallel.



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

Figure 5.10. Three images from the start of sequence of wave paddle oscillations. As the paddle resolution increases, different types of waves can be created.

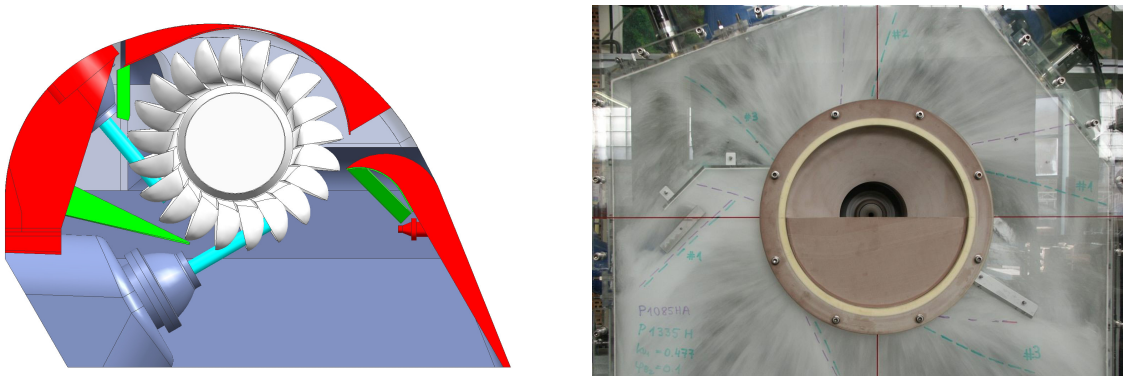
The wave maker module represented in figure 5.9 can be generated in parallel, and each piece is written to the DSM in parallel, which can be particularly useful when the number of polygons produced becomes very high. Paddle resolution/count may be dynamically adjusted and each paddle may be controlled by a frequency, phase and amplitude which may be linked to sliders in the GUI, or to the animation panel provided by ParaView (see figure 5.5). Patterns of motion may therefore be created. The engineer may adjust controls on the fly, but certain constraints must be imposed to prevent overlying large changes in positions of the paddle surfaces between time steps which can cause the simulation to fail. As shown in figure 5.10, during a short simulation run, the paddle is only allowed to move using a low amplitude but resolution can be increased on demand so that different wave types can be experimented. Note that at this stage the simulation does not support well re-meshing of deformable objects and more work is required to handle this type of re-meshing operation.

A considerable amount of effort has gone into making the use of the ICARUS plugin as easy as possible for an engineer to create a custom panel with GUI elements that can be adjusted to control the simulation, without any knowledge of ParaView or its internals, simply by creating XML to describe outputs, controls and inputs back to the simulation. There are times however, when it is necessary to create a custom control for a specific simulation such as a wave maker module and its associated interface, which are currently being developed.

5.2. Integrating ICARUS into an ALE-SPH code

In this section we focus on another code used for the simulation of water flows inside Pelton turbines. The Pelton turbine (see figure 5.11) is a hydraulic machine consisting of *buckets* on a rotating runner driven by high-velocity impinging water jets, used to produce energy from high head waterfalls. Numerical simulation can help in improving the runner design and the turbine efficiency, but traditional techniques are unable to accurately model the water sheets released from the buckets, because of excessive numerical diffusion. Using SPH, it is possible to study the interactions of water sheets issued from one jet on the other and the resulting possible perturbations.

The SPH-ALE code used [49] is a well-known SPH code developed by ANDRITZ HYDRO and ECL (Ecole Centrale de Lyon). The version of the code makes use of GPUs for accelerating the computation. However at the moment, the code is only capable of running on one single GPU (the multi-GPU version being still in development), therefore only a small number of particles (about 2×10^5 in the following examples) are used. While several models and test cases can be simulated using this code, we focus in the following sections on the Pelton runner simulation and see how the integration of our platform into this code can allow an engineer to perform real-time



(a) Two-jet horizontal Pelton turbine in its casing. **(b)** Horizontal Pelton test rig in operation.

Figure 5.11. Pelton turbine.

studies. In 5.2.1 we detail the in-situ visualization aspects and in 5.2.2 the steering aspects. We conclude on the integration of our framework into this simulation in 5.2.3. Also note that complex data array exchanges (used for object re-meshing), which have been highlighted in the previous simulation code, may only be integrated into this code in a future work presented in 5.2.3.

5.2.1. In-situ Visualization

An overview of the code (C++) is given in annex D.2. The code follows the usual structure of simulation codes with a phase of initialization and domain decomposition, and a computation loop over time. The specificity of this code is the strong usage of C++ template functions that allow the same code to be used with different data types. At the end of each iteration, data output is written to disk using the HDF5 format. Note that once the code was extended to make use of the HDF5 file format for its data output, using it with the `dsm` HDF5 driver was straightforward (as, again, switching drivers only requires one line of code to be modified in the simulation code).

On the post-processing side, ParaView is instantiated and the ICARUS plug-in loaded. When the user enables the DSM (which gets created on the ParaView server, see 3.4.2), a port is opened on the host, and the DSM service starts waiting for new connections. It also creates a configuration file that contains the host name, port, and inter-communication method (socket, MPI, DMAPP) that the simulation has to use for communicating with the DSM. When the simulation is launched, it reads this configuration file and it can then start sending data, etc, to the DSM.

The hierarchical data representation used by this code is defined in figure 5.12. In this case, the Pelton turbine is composed of 21 buckets, which are modeled using particles (but could equally well be modeled using meshes). These buckets are stored into separate HDF5 groups, named

Solid#0, Solid#1, …, Solid#20 respectively (which contain position, pressure, velocity arrays, etc). The fluid injected into the turbine is stored into a separate group called Particles#0 (to follow the H5Part layout and make eventually use of the H5Part reader presented in 4.2.2 for optimized reading).

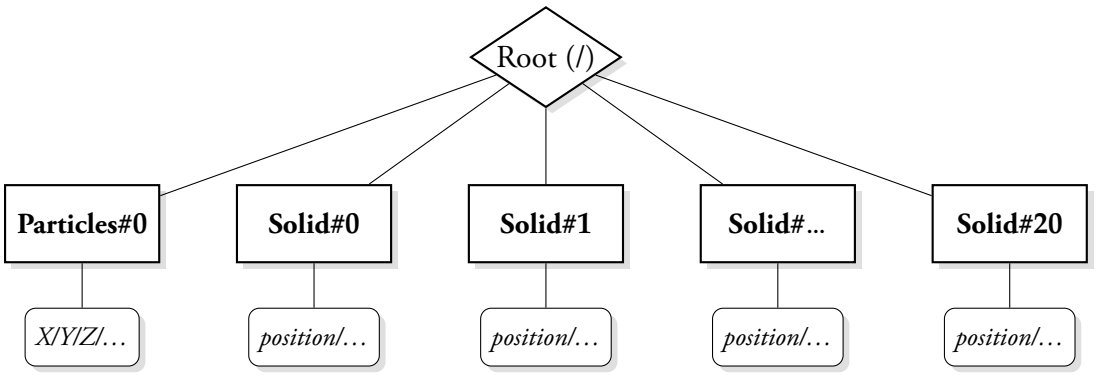


Figure 5.12. Hierarchical approach representation of SPH-ALE simulation data (for clarity, other attribute arrays such as pressure, velocity, etc, are not represented but are stored at the same level as the geometry arrays).

Note that in this case, as fluid particles are injected into the runner, the number of particles at a given time is undefined. While the XDMF reader requires by default the size of data to be explicitly stated, in our case by making use of the XML template format that we defined, dimensions do not need to be passed directly by the user, which is a very important condition for in-situ visualization. A shortened template file for reading the turbine data is given below:

```
<Visualization>
  <Xdmf>
    <Grid Name="Particles">
      <Topology TopologyType="Polyvertex"/>
      <Geometry GeometryType="X_Y_Z">
        <DataItem Name="X">/Particles#0/X</DataItem>
        <DataItem Name="Y">/Particles#0/Y</DataItem>
        <DataItem Name="Z">/Particles#0/Z</DataItem>
      </Geometry>
    </Grid>
    <Grid Name="Solid 0">
      <Topology TopologyType="Polyvertex"></Topology>
      <Geometry GeometryType="XYZ">
        <DataItem>/Solid#0/position</DataItem>
      </Geometry>
    </Grid>
    <Grid Name="Solid 1"> ... </Grid>
    ...
    <Grid Name="Solid 20"> ... </Grid>
```

```
  </Xdmf>
</Visualization>
```

As described before, all the different sub-blocks are included and written into XDMF *Grid* objects, one *Grid* for each type (in this case) of solid or fluid particles. When passing this XML template to the ICARUS plug-in, the XDMF reader will read data in parallel by grid (see 4.2.2). As illustrated in figure 5.13, which shows data redistribution using a simple *Process Id* filter, solid particles are evenly distributed between processes. However if we follow the previous XML template, fluid particle data can only be read by a single process. As the number of fluid particles being injected increases, being able to read the fluid particles in parallel from the DSM is also an important requirement. To ensure this point, we therefore make use of the same H5Part reader presented in 4.2.2 to obtain the result of figure 5.13.

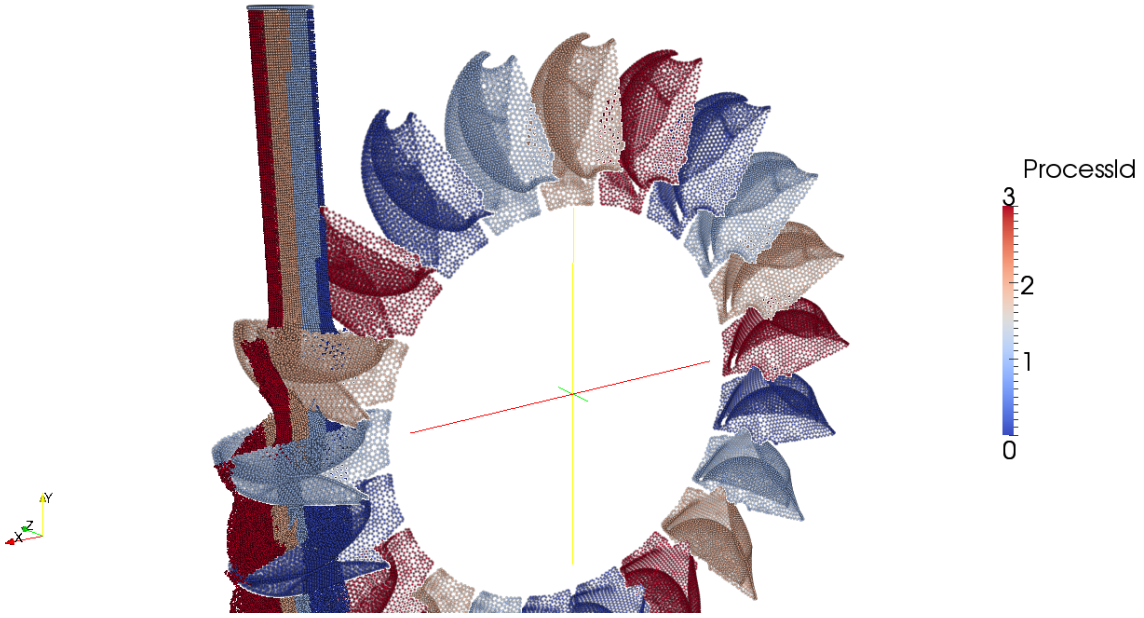
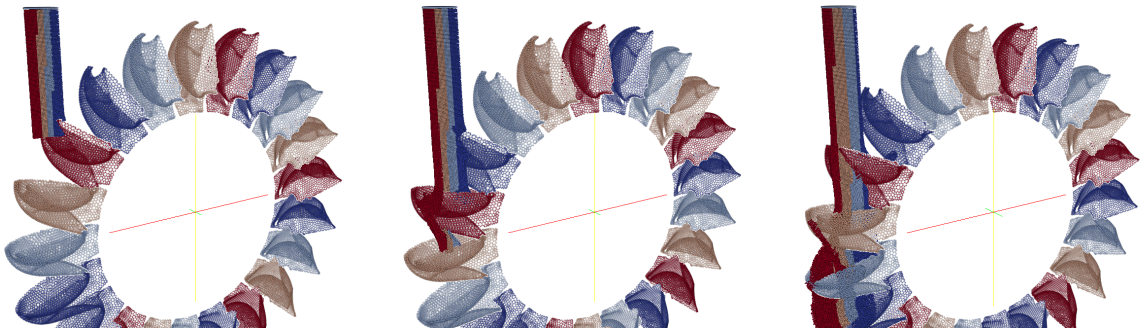


Figure 5.13. Pelton runner visualized in-situ using ICARUS. The solids are composed of $\simeq 5 \times 10^4$ particles while the fluid is composed of $\simeq 1.5 \times 10^5$ particles.

Being able to read all the data in parallel allows the analysis filters applied to the data to also operate in parallel without any extra data redistribution [4]. The outputs of probes, plots, bar charts all update automatically when the plugin has loaded new data from the DSM and triggers pipeline updates in the ParaView GUI. Owing to the fact that a separate DSM service thread is operating, data is being received even while the user is interacting with the previous result and the refresh of data happens without any user intervention. In a case like this where the code runs on GPU, a time step can be computed in sub-second time while the analysis may be performed in the same order of magnitude (or more depending on the analysis that is to be applied). Therefore being



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

Figure 5.14. Three images from a sequence where water is injected into the Pelton runner. Visualization is realized on 4 processes.

able to minimize as much as possible the I/O operations from the DSM is an important factor to guarantee a live and interactive visualization of data, without slowing the simulation code if the data output frequency is very high.

5.2.2. Computational Steering

The on-line steering of such a computation by using ICARUS can include variation of numerous upstream parameters in order to simulate different operating modes of the turbine in one simulation run. We consider in the following section the steering of simple parameters such as the modification of the runner velocity or the radius of the jet injecting fluid particles into the turbine.

Defining Steering Commands

To create commands in the ICARUS plug-in, users must extend the XML description template (used for reading data) with an `Interaction` section (see 4.2.3). To modify parameters such as the runner velocity or the jet radius, the user only needs to send scalar values (from the plug-in) back to the DSM. In this particular case, one may define `DoubleVectorProperty` fields in the XML description and specify a range domain, which is a constraint to the parameter values that the simulation cannot exceed. We consider the following template:

```
<Icarus>
  <Visualization>
    ...
  </Visualization>
  <Interaction>
    <DoubleVectorProperty
```

```
      name="NewRunnerVelocity"
      command="SetSteeringValueDouble"
      label="Runner velocity"
      number_of_elements="1"
      default_values="-62.0" >
      <DoubleRangeDomain name="range" min="-105.0" max="10.0"/>
      <Documentation>
        Set the rotational velocity
      </Documentation>
    </DoubleVectorProperty>
    <DoubleVectorProperty
      name="NewJetDiameter"
      command="SetSteeringValueDouble"
      label="Jet radius"
      number_of_elements="1"
      default_values="1.0" >
      <DoubleRangeDomain name="range" min="0.0" max="1.0"/>
      <Documentation>
      Set the jet radius
      </Documentation>
    </DoubleVectorProperty>
  </Interaction>
</Icarus>
```

The velocity values can be here positive or negative, leading the runner to rotate in one direction or the other and setting a value of 0 will even stop the rotation of the turbine. The jet radius can be reduced proportionally to its initial size and can therefore even be reduced to 0, stopping the injection of particle fluids into the turbine.

Modifying the Simulation

Without any modification of this simulation, computing a time step using the GPU with a low number of particles (2×10^5) and single precision can be achieved in 1×10^{-1} seconds[1]. Therefore, adding commands and steering operations without slowing down the simulation is an important requirement. However as we showed in 4.2.3, checking for commands and new data from the DSM can be performed in a much lower order of magnitude (i.e., in the order of the millisecond). As described in annex D.2, the steering calls inserted follow for this code the same scheme adopted for reloading meshes into SPH-flow and we check for new values using a call to `h5fd_dsm_steering_is_set` with the corresponding name of the steered parameter (e.g.,

[1] The time highly depends on the precision used, on the fluid/structure interaction computations and on various solving parameters that can be set in the simulation to control the precision of the results.

`NewJetDiameter`) at the beginning of an iteration. If new values are found in the DSM, the requested parameters are modified.

While we are looking in this particular case for a very interactive simulation that can be steered (as opposed to the SPH-flow test case running on CPUs), adding several analysis and rendering filters may actually slow down the simulation if the frequency of the outputs is too high. Therefore the output frequency parameter could also be interactively steered by the user to adapt the visualization and steering experience to his needs. Also, while writing a time step from the simulation to the DSM does not require waiting for the actual post-processing operations to complete, if the post-processing side has locked the file (see 3.1.4) and requires access to the DSM multiple times during the update of the ParaView pipeline, the simulation will have to wait for the file to be unlocked before being able to write a time step and carry on computation. In other words, making the post-processing application keep the file lock can stop the simulation from writing data. One solution that we will implement in a further implementation of the `dsm` driver will be to query the status of the lock, so that one can potentially skip the writing of a time step if necessary.

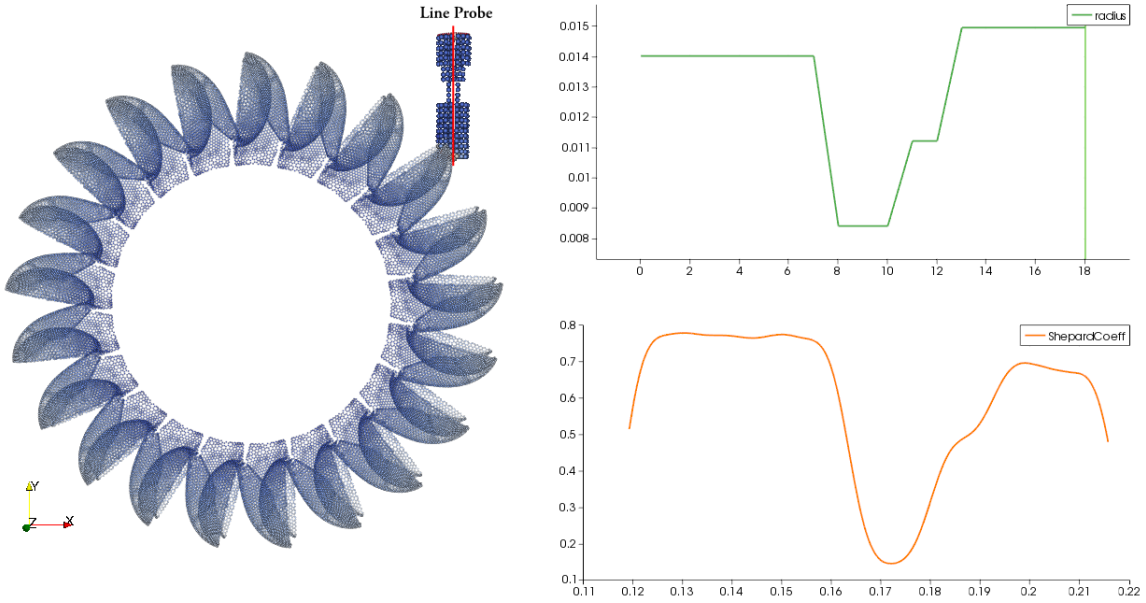


Figure 5.15. Jet radius steered over time using ICARUS.

Figure 5.15 shows an example of a steering session using ParaView analysis filters. In this example, the jet radius is varying over time and one can easily notice that the injection of particles decreases when the radius decreases. On the right of figure 5.15 are displayed the actual jet radius in function of time and the corresponding Shepard coefficient [30]. The Shepard's Method can be defined as an inverse distance weighted calculation, which sums contributions from all neighboring particles to the sample point. The Shepard coefficient has the useful property that within the

material (when using normalized units), it will be close to unity, outside it falls to zero, and at the boundary of the fluid takes on values in between (which is then particularly useful for applying contour filters). In this case, the Shepard coefficient is obtained in ParaView using an SPH line probe filter [9] (represented in red for clarity), which has been placed in the injected fluid along the Y-axis. As the jet width is particularly small, placing this line probe filter along the Y-axis allows us to get values of the Shepard coefficient varying at the same time of the jet radius (as the number of neighboring particles is small).

As described in annex D.2, other parameters such as the runner velocity, the deflector angles or the bucket position can be steered using the same method. These parameters can also be steered using the animation panel of ParaView so that a parameter value given by the user can be incrementally reached over time (see 5.1.2).

5.2.3. Conclusion and Future Developments

In the previous section, we showed how one can make use of the ICARUS framework to modify parameters of a very interactive simulation on-the-fly. Data can be analyzed by adding analysis filters to the ParaView post-processing pipeline, these filters being updated as new data is picked up from the DSM. In a real scenario such as the one we presented, an engineer can enable interactive modification of a parameter such as the jet radius or the runner velocity with little effort (i.e., by simply extending an XML description file and adding the corresponding calls in the simulation to get the data).

However, while we showed that constraints on the parameters can be defined by setting a *range* domain, we are not able at the moment to define *interactive* constraints, which would for example prevent excessive values from being passed to the system (which would in turn break physics laws). Even though one can set animation keyframes from ParaView, allowing parameters to be increased from a limited value over time (by following a ramp/step/... interpolation function), these parameters are not directly linked to the simulation results and are only *user* controlled. Therefore to go further in our steering approach, additional developments are necessary on the post-processing side to allow direct interaction and control between the simulation and the GUI entered parameters. When modifying a parameter, the simulation must therefore, for now, compute a value that is said *acceptable* for the system, which is of the responsibility of the user or of the simulation code developer.

It is worth noting that the test case previously presented only realizes at the moment exchanges of parameters and does not make use of all the features provided by our framework. Further developments may therefore include re-meshing capabilities such as the ones presented in section 5.1.2.

While the previous steering requests are only related to the modification of turbine parameters, another problem, which could be integrated into the simulation, is the erosion impact from sediments onto the runner buckets (see figure 5.16).

Figure 5.16. Damages on Pelton runner, Malana, India (credit: ANDRITZ HYDRO).

Based on the jet pressure values and the concentration of sediment in the water, one would apply a Paraview *erosion* filter onto each bucket (which has direct interaction with the water jet) and generate deformed buckets (that include erosion results). Note that in this simulation buckets are composed of particles; therefore these buckets must be composed of a reasonably high number of particles (so that interpolating a particle displacement due to erosion can actually make sense). Each of the eroded/modified buckets would then be sent back to the DSM and reloaded into the simulation (following the scheme of figure 5.6). The simulation would therefore include in the results another non-negligible factor that can directly impact the Pelton runner; this would define another scenario that can directly help an engineer in solving another complex type of situation, the erosion effects being monitored and tweaked using different controls such as the sediment concentration, etc.

5.3. NextMuSE Objectives and Validation

We showed in the previous examples that making use of the ICARUS interface and of the HDF5 `dsm` driver requires little effort for an engineer before being able to visualize and interact with a simulation (as only a few lines of codes need to be modified). Moreover if the simulation code does not make use of HDF5 already, making this effort will also improve in most cases simulation I/Os (as HDF5 provides one of the most comprehensive and efficient way of doing I/Os from a simulation) and this will therefore be of some benefit for the simulation.

One of the first objectives from the NextMuSE project was to create an interface capable of processing data at an estimated rate of about 10×10^6 particles per second. We showed in 5.1.1 that we are in fact capable of processing 20×10^6 particles in sub-second time (which can actually be much lower as this time includes memory allocations specific to the SPH-flow code). Note that this data rate is obtained on a high-bandwidth network and therefore would be much lower on low-bandwidth networks such as Ethernet networks. The two SPH codes that we presented had their output modified in order to write out data using the HDF5 format; they operate in this case as if they were using a normal parallel file system, but instead have traffic re-routed directly to the post-processing nodes hosting the DSM.

On the post-processing side, when data is read back from the DSM, we showed in 5.2.1 that all the different steps of the visualization, analysis operations can be performed in parallel. When new data is written into the DSM from the simulation, notifications are triggered and all the visualization and analysis filters are automatically updated; all the ICARUS GUI and ParaView environment is multi-threaded so that user interactions may take place at the same time. When steering, we showed in 5.1.2 and in 5.2.2 that we are capable of sending back commands, by simply adding the appropriate sections into the XML description file that is passed to the plug-in, or more complex types of data such as meshes that can be used for moving or deforming an object in the simulation. For the specific needs of the application domains, specific GUI widgets can be created such as the one we presented in 5.1.3 to allow manual control of geometric elements. The generated elements can then be sent back to the DSM in parallel.

Integrating the ICARUS interface into the two simulation codes that we presented brought several advantages: one can visualize the time steps produced by the simulation without suffering from the bottleneck introduced by disk I/Os and it is now possible to interact with the simulation, modify objects on-the-fly and analyze the results. For SPH-flow, an engineer is able to interactively modify the paddle amplitude and frequency (and in a future version the paddle resolution) and generate and observe the behavior of different wave types onto an object placed in the tank. One can therefore reproduce experimental results observed in the (real) water tank and compare them with the simulation to improve the model. The object can also be modified on-the-fly and one can also imagine creating another specific GUI control for generation and modification of hull shapes (everything happening in parallel). For the SPH-ALE code presented, one can for now interactively tweak the Pelton runner parameters and observe their impact, using ParaView animation keyframes for automatic interpolation of the parameter values over time, while doing analysis of the results. In the future, the interface can make the simulation integrate new parameters, such as the erosion impact onto the Pelton buckets, by reloading through the DSM a new set of modified particles into the simulation.

The interface that we have created is very light and therefore additional parameters can be easily steered without significant effort. Integrating the interface into a simulation for in-situ visualization and steering can be summarized in three steps: make the code use of HDF5 and add a call to our driver; create an XML description file for reading data; define interactions in the simulation code and in the XML description file. The interface is therefore very adapted to the work of an engineer, and does not require any strong knowledge in programming or in computer science. Moreover as previously demonstrated, the interface provides good performance and scalability, data being transferred at a transfer rate of more than 60 GB/s on the Cray XK6 used for our tests.

Chapter 6.

Conclusion and Perspectives

Several layers of our approach have been studied in this thesis; we have shown how in-memory and distributed HDF5 files may be used to loosely couple simulation and post-processing nodes of a supercomputer to a GUI on the desktop, enabling interactive manipulation of the simulation. The ability to make use of a single description file, which enables the use of multiple customized readers that have been tuned for a particular kind of hierarchical data layout demonstrates the flexibility of the approach and the ease with which it may be extended. By fine tuning data transfers using one-sided approaches as well as the simulation and analysis I/Os, we have minimized the delay experienced by the user before performing analysis tasks on their data, bypassing the disk I/O bottleneck introduced by the traditional visualization approach. In the following sections, we present some of the future directions that can improve and remove the current restrictions imposed by our framework.

6.1. Using a PGAS Model

All the results presented in this thesis appear to be typical for the kinds of system tested, but can however be affected by the network topology and capabilities, system configuration, number of nodes used, number of processes per node, and so on. The space of potential combinations of parameters for plots is beyond what can be presented, so certain decisions as the number of processes to use per node were made to try to maximize the data injection and network saturation to give representative results.

The improvements in transfer rates found when using redistribution are broadly in line with expectations. In fact, the advantages of data redistribution are well known and date back to the origins of message passing [75]. Many projects have made use of block-cyclic distribution as a means of improving performance for scattered data. In particular PGAS languages [1] provide

options for shared array allocation using block cyclic layouts, which can improve algorithmic performance. Our flexible communicator design opens up the possibility that a PGAS based layer could be used directly instead of MPI or DMAPP as we have presented here and we shall pursue this in future work.

6.2. Towards a Virtual Object Layer

In our approach, we focused on redistribution strategies (see 3.2) that operate at the memory level. These redistribution strategies allow transfers to be optimized by making use of all the DSM nodes available. However when data is received by the DSM, memory copies remain between the DSM hosts and the ParaView servers, even if they do share the same nodes. In fact the `dsm` HDF5 driver that we implement does not allow DSM memory pointers to be externally re-used and therefore imposes an extra memory copy between DSM buffers and VTK objects; this is because we operate at the memory level (*Virtual File Layer*) and not at the object level.

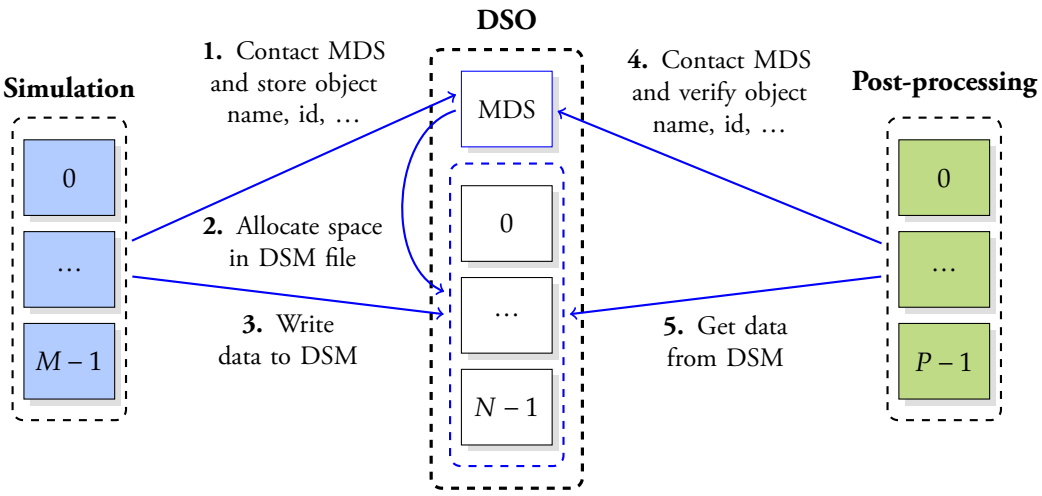


Figure 6.1. The distributed shared object approach can extend the current DSM approach by operating at the HDF object level.

Future versions of HDF5 can allow us to remove these extra memory copies by implementing a *Virtual Object Layer* (VOL). Using the VOL, we can operate from a higher level and map HDF objects to any memory representation. As described in figure 6.1, we can extend the DSM approach by storing additional object information; this approach can thus be defined as a *distributed shared object* approach. In this case, a node may host a metadata server (MDS), which can store object information and map the object to the DSM. The object is then written into the DSM using the same mechanisms implemented in the approach presented in this thesis, taking advantage of

the redistribution strategies and modes of communication that we integrated. Consequently, more advanced redistribution strategies (operating at the object level) could also be defined, mapping particular objects to specific nodes.

Making use of this upper layer could therefore improve several aspects of the work that was presented in this thesis; it would allow us to define a more complex system where objects can be dynamically allocated and accessed without any extra memory copy, hence removing the main limitation, from which our framework can currently suffer.

6.3. Storing Multiple Files

The DSM approach that we presented only makes use of a single file for the exchange of data and, as we saw in 3.3.4, every new in-memory file written into the DSM replaces the file that is currently stored (i.e., only one simulation time step can be stored at the same time in the DSM). This is another limitation and being able to store multiple files in the DSM could therefore add more flexibility to our approach.

Two main reasons can be given for adding this capability and the first reason is that being able to store multiple files in the DSM means that multiple time steps can actually be stored, giving the opportunity for the simulation to be rerun from a determined time step. If for any reason during a steered simulation run, a user needs to modify parameters at a given time step, the simulation could be relaunched from that particular time step. In other words, a user could write a time step into the DSM, steer parameters, write another time step, visualize the results and if the results were wrong or were not satisfying some defined criteria, the user would still be able to go back to the previous time step and modify the parameter he has just changed.

The second reason for adding this capability is that steering commands (used for modification of user defined parameters) are for now stored in the same DSM file (see 3.3.3) (while commands such as *pause* and *resume* are hidden from the user by the driver implementation). Since these commands are written into the same file, there is a potential risk for the user to lose these commands; if (see 3.3.4), for example, a user writes a command from the GUI, but checks for these commands only after the simulation has written a time step, in which case the file will have been wiped and the steering command will have been lost. One might argue that users could check for these commands every time before re-creation of the file by the simulation, however this would not prevent commands from being lost in more complex scenarios where they must be picked up at a particular time step (and have been written several time steps earlier). Adding this capability would therefore remove the constraint where users have to always check for these commands before they create a new file in the DSM.

6.4. Conclusion

Combination of the three previous future directions can greatly improve the flexibility of the approach by making use of: PGAS models for redistributing data among the DSM processes; the virtual object layer for manipulating HDF objects and avoid extra memory copies when these objects have to be mapped to different memory structures; multiple files to store multiple time steps of the simulation and separate steering commands from simulation data. While these developments are low-level developments, they would have a direct impact on the steering and visualization session of a simulation code: making use of advanced redistribution strategies can increase the processing speed of user data; decreasing memory copies can concretely mean that larger amounts of data can be visualized and accessed; and finally, storing multiple time steps implies more possibilities of steering without having to restart the simulation, which can be seen as a *replay* capability.

It is also worth noting that at the moment, the limiting factor that prevents *real-time* control of the simulation is usually the simulation speed itself, which is an order of magnitude away from where it should be to *feel* truly interactive. We are for example able to control a simulation of 0.5 billion particles on 2048 (or more) cores; however, reaching this precision means that the corresponding physical time step of the simulation must be reduced accordingly, which can then become too small to allow the user to perform intuitive operations such as moving objects freehand on-the-fly (using a standard controller such as a mouse, joystick or haptic controller). When reaching very high precision and resolution, a latency must therefore be expected before controls have real impact on the simulation. Consequently improving the simulation so that it can give real-time response must still be one of the primary objectives of a simulation developer who wants to have the most interactive session possible, though most of the time compromises will have to be found between the precision (number of particles, mesh resolution) and the level of interactivity requested.

Annex A.

Cray Gemini Interconnect and uGNI-Based Communicator

Introduced as the successor of Seastar [11], the Gemini interconnect [6] is a router chip for Cray supercomputer systems that provides low latency and very high bandwidth. It is designed to provide high performance for global address space programming and enables efficient implementation of programming languages such as SHMEM [25] and PGAS compilers [1] (UPC, and Co-array Fortran). Two user interfaces are provided, DMAPP and uGNI. While the first one is designed for distributed memory programming and one-sided accesses, the second one allows specific communication domains to be established and more control over the different Gemini communication mechanisms. As mentioned in 4.1.3, to optimize our communication interface, we developed a low-level interface that makes use of DMAPP, which allows high throughput but limits us to a static connection mechanism (see 4.1.3). In this appendix, we show more advanced properties of the Gemini interconnect and an additional inter-communicator implementation that makes directly use of the lower level API called *uGNI* (User level Gemini Network Interface) [62].

A.1. Architecture

To understand how our communicators operate, it is necessary to understand some of the Gemini internal architecture. Each Gemini card provides two *network interface controllers* (NICs). Each of the NICs can support two nodes (as illustrated in figure A.1): the x and z dimensions have double links, one corresponding to each node; the y dimension has a single link to the neighboring router chip since the two nodes within the router can be considered connected across the y axis. Traffic between two nodes connected to a single Gemini is routed internally. Gemini can also perform adaptive routing, distributing traffic over lightly loaded links (multiple paths can be defined by making use of the multiple NIC links).

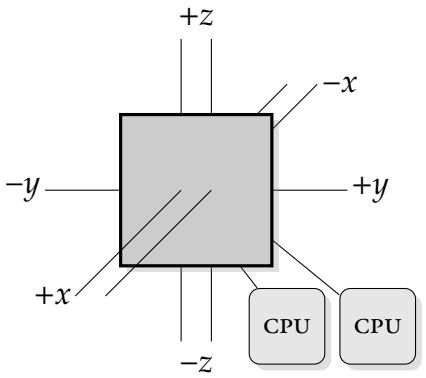


Figure A.1. The Gemini NIC can support two nodes and has double links in x and z directions.

A.1.1. One-sided Transfers

Gemini transfers data in a one-sided manner without any OS intervention. As we showed in 3.1.3, to be able to transfer data to a remote target, the remote virtual addresses, source addresses and size of the data that needs to be transferred must be specified.

Transfer Mechanisms

There are two mechanisms for accessing remote memory on another node: *Fast Memory Access* (FMA), and *Block Transfer Engine* (BTE). The first one is dedicated to small messages while the other one is dedicated to large messages. Note that more time is required to set up data for a BTE transfer, than for an FMA transfer, but once initiated, there is no further involvement by the processor core. For some transfer operations, the GNI kernel driver first exchanges datagrams, which contain messaging parameters, to initialize communication between processes.

As these mechanisms are one-sided, a notification mechanism must be used to track progress of one-sided requests; this is handled through *Completion Queues* (CQ). For brevity, we do not detail here the creation of completion queues and their association to logical end points that have been defined within the communication domain, more details can be found in the Cray uGNI documentation.

Address Translation Mechanisms

Two hardware mechanisms can then be used by the Gemini NIC to translate virtual to physical memory references (N.B.: the following is mainly taken *as is* from the Cray documentation):

1. The *Graphics Address Remapping Table* (GART) is a feature of many AMD64 processors that allows access to virtually contiguous user pages that are backed by non-contiguous physical pages. The GART mechanism aggregates the Linux standard 4 kB pages into larger virtually contiguous memory regions. The contiguous pages exist in a portion of the physical address space known as the *Graphics Aperture*. The GART's graphics aperture size is 2 GB. Therefore, the total memory which can be referenced through GART is limited to 2 GB per compute node.

2. The *Memory Relocation Table* (MRT) on the Gemini NIC maps the memory references contained in incoming network packets to physical memory on the local node. Memory references through the MRT map to a much larger address range than they do through the GART. Each NIC has its own MRT. MRT page sizes range from 128 kB to 1 GB, but all the entries on a given node must have the same page size. The MRT entries are created by kGNI (Kernel level Gemini Network Interface) in response to requests from the application, usually the uGNI library. There are 16 K MRT entries. The default MRT page size is 2 MB, which maps to 32 GB (16K $\times$ 2 MB).

Depending on the size of the allocated memory region and other default behavior, the memory registration function (of uGNI/DMAPP) asks the kernel to create either GART entries on the AMD processor, or, in the case of huge pages, create entries in the Memory Relocation Table (MRT) on the NIC, to span the allocated memory region. The second mechanism is usually preferred in our case as large memory pages can therefore be mapped to the memory of our DSM (and large data transfers can be performed).

Memory Allocation and Registration

User virtual memory that has to be read or written across nodes, must be generally first registered on the node; its physical location and extent loaded into the Gemini *Memory Descriptor Table* (MDD) and either the Opteron GART or the Gemini MRT.

Note that when using DMAPP, two different memory allocation mechanisms can be used: from the *symmetric* heap or from the *private* heap (using a normal `malloc`). DMAPP `mmaps` the symmetric heap directly, regardless of its size, to the hugetlbfs file system if it is mounted. When an application's memory requirements exceeds the GART aperture size (2 GB) on a single node, the application must be linked with the `libhugetlbfs` library, to use the larger address range available with huge pages (again this is generally the behavior one will encounter most of the time when using the DSM, therefore the application must always be linked to the `libhugetlbfs` library when using the DMAPP inter-communicator). Once user memory registered to the node and memory descriptors exchanged between the sending and the receiving process, transactions

can be issued in a one-sided manner to the remote memory without further involvement of the remote process.

A.1.2. uGNI Microbenchmark

To illustrate the two different mechanisms, BTE and FMA, we make use here of the same Cray Gemini system used in 4. The two main mechanisms, `GNI_PostFma` (FMA, medium size messages) and `GNI_PostRdma` (BTE, large size messages), are represented in figure A.2. From this figure, one can determine a *good* offload switch point between FMA and BTE mechanisms, which would be here of 16 kB. But note that this point also depends on other parameters, such as the latency (not represented here), etc. It is also worth noting that we can almost reach the theoretical peak bandwidth of 7 GB/s using the BTE mechanism when transferring raw data between the two nodes.

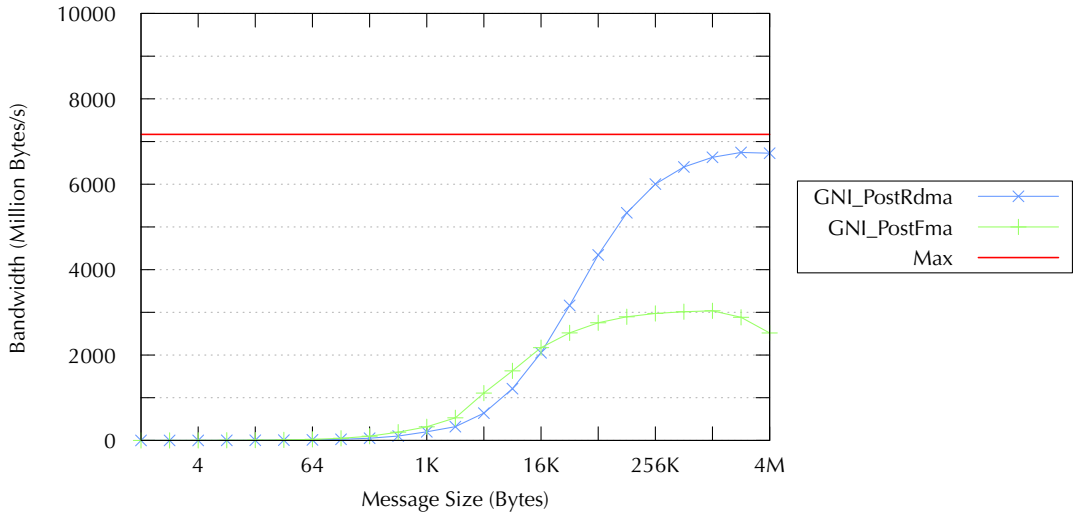


Figure A.2. Inter-node micro-benchmark using uGNI (blocking) put operations. FMA short messaging (SMSG) is not represented here and can only be used if the message size does not exceed 64 kB.

A.2. uGNI-Based DSM Communicator

There are two main motivations for implementing a uGNI communicator into the DSM: the first one is to be able to finely control which offload mechanisms and techniques will be used; the second one is to restore missing MPI dynamic connection support. As mentioned in section 4.1.3, simply making use of MPI restricts the usage of the DSM to only launching both sender and receiver as

part of the same job (MPMD). The main reason for this is that Cray ALPS (the Application Level Placement Scheduler) restricts the use of only one single *protection tag* and *cookie* per job, which prevent two different applications from establishing communication domains (and therefore interacting) with each other (which would be seen as a security issue). This model is referred as *private protection domain* model.

However, using the lower level uGNI interface, *flexible communication domains* have also now been introduced, which can allow one to dynamically connect different jobs (that share the same pre-existing protection domain) together. Applications or sets of applications may therefore use one of the following protection models (N.B.: the following is mainly taken *as is* from the Cray documentation):

1. Multiple cooperating applications owned by the same user may share a communication domain. This prevents non-cooperating applications from accessing another application's memory regions. A communication domain that is shared between cooperating applications is also referred to as a *shared protection domain*.

2. Multiple users of several applications may share the use of a system service, which uses a *system dedicated protection domain*.

In our case, we can make use of user-defined shared protection domains. Processes connected together are then accessed using our own network translation table that maps processes IDs to target addresses.

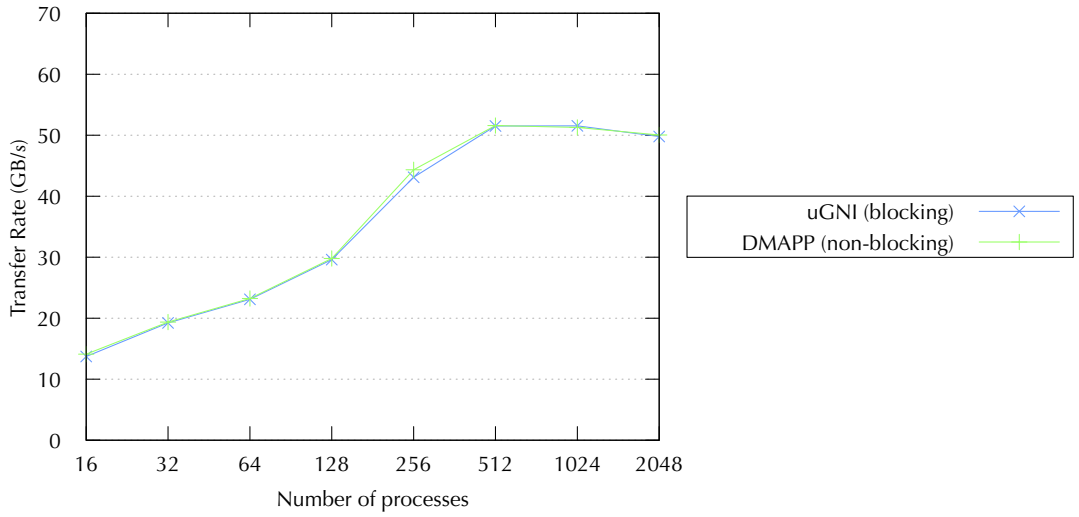


Figure A.3. Write transfer rate of an (in-memory) HDF5 file composed of one single dataset using uGNI and a contiguous distribution—DSM size of 40 GB and distributed among 160 processes (40 nodes).

Once communication established, one can set up a DSM server on the machine and have the simulation dynamically launched, connected (and eventually restarted) reading and writing data from/to the DSM. This is very important for our approach as one can therefore launch the DSM as a service and use it as a temporary storage, dynamically connecting and disconnecting from it. Figure A.3 shows early results obtained using this communicator. The same configuration of 4.1.3 is used. Note that the performance between the DMAPP and uGNI communicators is very similar in this scenario where data is simply sent to the DSM. More testing and optimization would however be required in this case to validate our approach and make a more reasonable comparison with DMAPP.

Although this communicator is still in development, we also aim at using it in the future as the default communicator for DSM exchanges (on Cray platforms that feature Gemini interconnects and later versions).

Annex B.

Server API

The H5FDdsmManager class presented below is taken out from the `H5FDdsm` library available at `https://hpcforge.org/projects/h5fddsm`. This class aims at providing basic server configuration for hosting and accessing a DSM buffer object [1].

H5FDdsmManager Class

```
class H5FDdsm_EXPORT H5FDdsmManager : public H5FDdsmObject
{
  public:
     H5FDdsmManager();
    ~H5FDdsmManager();

    // Description:
    // Get the process rank and communicator size.
    H5FDdsmGetValueMacro(UpdatePiece, H5FDdsmInt32);
    H5FDdsmGetValueMacro(UpdateNumPieces, H5FDdsmInt32);

    // Description:
    // Set/Get the MPI Communicator used by this DSM manager.
    void SetMpiComm(MPI_Comm comm);
    H5FDdsmGetValueMacro(MpiComm, MPI_Comm);

    // Description:
    // Get the DSM buffer used by this DSM manager.
    H5FDdsmGetValueMacro(DsmBuffer, H5FDdsmBufferService*);


```

[1] While this interface is C++ only, we also aim at providing in future versions of the library a C interface that will be able to perform the same type of operations.

```
    // Description:
    // Set/Get the size of the buffer to be reserved on this process
    // the DSM total size will be the sum of the local sizes from all
    // processes.
    H5FDdsmSetValueMacro(LocalBufferSizeMBytes, H5FDdsmUInt32);
    H5FDdsmGetValueMacro(LocalBufferSizeMBytes, H5FDdsmUInt32);

    // Is the DSMBuffer auto allocated within the driver or not.
    H5FDdsmGetValueMacro(IsAutoAllocated, H5FDdsmBoolean);
    H5FDdsmSetValueMacro(IsAutoAllocated, H5FDdsmBoolean);

    // Description:
    // Set/Get IsServer info.
    H5FDdsmSetValueMacro(IsServer, H5FDdsmBoolean);
    H5FDdsmGetValueMacro(IsServer, H5FDdsmBoolean);

    // Description:
    // Set/Get IsStandAlone info.
    H5FDdsmSetValueMacro(IsStandAlone, H5FDdsmBoolean);
    H5FDdsmGetValueMacro(IsStandAlone, H5FDdsmBoolean);

    // Description:
    // Set/Get IsDriverSerial info.
    H5FDdsmSetValueMacro(IsDriverSerial, H5FDdsmBoolean);
    H5FDdsmGetValueMacro(IsDriverSerial, H5FDdsmBoolean);

    // Description:
    // Set/Get the interprocess communication subsystem
    // Valid values are H5FD_DSM_TYPE_UNIFORM, H5FD_DSM_TYPE_BLOCK_CYCLIC.
    H5FDdsmSetValueMacro(DsmType, H5FDdsmInt32);
    H5FDdsmGetValueMacro(DsmType, H5FDdsmInt32);

    // Description:
    // Set/Get the DSM block length when using H5FD_DSM_TYPE_BLOCK_CYCLIC.
    H5FDdsmSetValueMacro(BlockLength, H5FDdsmUInt64);
    H5FDdsmGetValueMacro(BlockLength, H5FDdsmUInt64);

    // Description:
    // Set/Get the interprocess communication subsystem
    // Valid values are: - H5FD_DSM_COMM_MPI
    //                   - H5FD_DSM_COMM_SOCKET
    //                   - H5FD_DSM_COMM_MPI_RMA
    //                   - H5FD_DSM_COMM_DMAPP
    H5FDdsmSetValueMacro(InterCommType, H5FDdsmInt32);
    H5FDdsmGetValueMacro(InterCommType, H5FDdsmInt32);
```

```
67
68      // Description:
69      // Set/Get UseStaticInterComm -- Force to use static MPI comm model
70      // when dynamic MPI communication is not supported by the system.
71      H5FDdsmSetValueMacro(UseStaticInterComm, H5FDdsmBoolean);
72      H5FDdsmGetValueMacro(UseStaticInterComm, H5FDdsmBoolean);
73
74      // Description:
75      // Set/Get the published host name of our connection.
76      // Real value valid after a Publish call has been made.
77      H5FDdsmSetStringMacro(ServerHostName);
78      H5FDdsmGetStringMacro(ServerHostName);
79
80      // Description:
81      // Set/Get the published port of our connection.
82      // Real value valid after a Publish call has been made.
83      H5FDdsmSetValueMacro(ServerPort, H5FDdsmInt32);
84      H5FDdsmGetValueMacro(ServerPort, H5FDdsmInt32);
85
86      // Description:
87      // Wait for a connection
88      // Only valid after a Publish call has been made.
89      H5FDdsmBoolean GetIsConnected();
90      H5FDdsmInt32 WaitForConnection();
91
92      // Description:
93      // Wait for a notification - notifications are used to trigger user
94      // defined tasks and are usually sent once the file has been closed
95      // but can also be sent on demand.
96      H5FDdsmBoolean GetIsNotified();
97      void ClearIsNotified();
98      H5FDdsmInt32 WaitForNotification();
99      void NotificationFinalize();
100
101     // Description:
102     // Get the notification flag - Only valid if GetIsNotified is true.
103     H5FDdsmInt32  GetNotification();
104     void ClearNotification();
105
106     // Description:
107     // Create a new DSM buffer of type DsmType using a local length of
108     // LocalBufferSizeMBytes and the given MpiComm.
109     // If using inter communicators, additional options may be specified
110     // such as ServerHostname, ServerPort, IsServer, InterCommType, etc.
111     H5FDdsmInt32 Create();
```

```
112
113     // Description:
114     // Destroy the current DSM buffer.
115     H5FDdsmInt32 Destroy();
116
117     // Description:
118     // Connect to a remote DSM manager (called by client).
119     H5FDdsmInt32 Connect(H5FDdsmBoolean persist = H5FD_DSM_FALSE);
120
121     // Description:
122     // Disconnect the manager from the remote end
123     // (called by client and server).
124     H5FDdsmInt32 Disconnect();
125
126     // Description:
127     // Make the DSM manager listen for new incoming connection
128     // (called by server).
129     H5FDdsmInt32 Publish();
130
131     // Description:
132     // Stop the listening service (called by server).
133     H5FDdsmInt32 Unpublish();
134
135     // Description:
136     // Open the DSM from all nodes of the parallel application,
137     // all nodes within a communicator must participate
138     // this function must be paired with a matching Close
139     // Use H5F_ACC_RDONLY for queries
140     // Use H5F_ACC_RDWR   for read/write.
141     H5FDdsmInt32 OpenDSM(H5FDdsmUInt32 mode);
142
143     // Description:
144     // Close the DSM from all nodes of the parallel application,
145     // all nodes within a communicator must participate
146     // this function must be paired with a matching Open.
147     H5FDdsmInt32 CloseDSM();
148
149     // Description:
150     // Returns 1 if OpenDSM has been called and the fapl and file handles
151     // have been cached. These are released when CloseDSM is called
152     // all nodes within a communicator must participate.
153     // Return 0 otherwise.
154     H5FDdsmBoolean IsOpenDSM();
155
156     // Description:
```

```
157     // If the DSM has been opened and handles cached, this returns the
158     // cached handle, otherwise H5I_BADID (-1). Use with great care.
159     hid_t GetCachedFileHandle();
160     hid_t GetCachedFileAccessHandle();
161
162     // Description:
163     // If the .dsm_config file exists in the standard location
164     // ${ENV{H5FD_DSM_CONFIG_PATH}/.dsm_config then the server/port/mode
165     // information can be read. This is for use the by a DSM client.
166     // DSM servers write their .dsm_config when Publish() is called
167     // Returns false if the .dsm_config file is not read.
168     H5FDdsmInt32 ReadConfigFile();
169
170     H5FDdsmInt32 WriteSteeredData();
171     H5FDdsmInt32 UpdateSteeredObjects();
172
173     // Description:
174     // Set/Get the current given steering command.
175     // The command is then passed to the simulation.
176     void SetSteeringCommand(H5FDdsmConstString cmd);
177
178     // Description:
179     // Set values and associated name for the corresponding interaction.
180     void SetSteeringValues(const char *name, int numberOfElements,
181       int *values);
182     H5FDdsmInt32 GetSteeringValues(const char *name, int numberOfElements,
183       int *values);
184
185     // Description:
186     // Set values and associated name for the corresponding interaction.
187     void SetSteeringValues(const char *name, int numberOfElements,
188       double *values);
189     H5FDdsmInt32 GetSteeringValues(const char *name, int numberOfElements,
190       double *values);
191
192     // Description:
193     // Return true if the Interactions group exists, false otherwise.
194     H5FDdsmInt32 GetInteractionsGroupPresent();
195
196     // Description:
197     // Set/Unset objects.
198     void SetDisabledObject(H5FDdsmConstString objectName);
199 };
```

Annex C.

Steering API

The H5FDdsmSteering interface presented below is taken out from the `H5FDdsm` library available at `https://hpcforge.org/projects/h5fddsm`. These routines aim at providing a comprehensive steering interface[1] for data and parameter exchange between a simulation code and the DSM (see section 3.3).

H5FDdsmSteering Interface

```
/* Description:
 * Initialize the steering interface. This must be called before using
 * the other steering functions.
 */
herr_t H5FD_dsm_steering_init(MPI_Comm intra_comm);

/* Description:
 * Refresh and update the state of raw steering commands such as
 * pause/resume and datasets that have been enabled/disabled through
 * the GUI. Any simulation code that needs to check for these commands
 * should make use of this call. In case of a pause command being sent
 * in, the simulation waits for a resume command at this control point.
 * Note that creating a new HDF file does not reset these values.
 */
herr_t H5FD_dsm_steering_update();

/* Description:
 * Test if a given dataset is enabled or not in the GUI.
 * Either the HDF path of a particular dataset or the grid object name
 * can be given (In this last case, it has to match the name given in
 * the template).
```

[1] The Fortran interface is also available but not presented here for clarity.

```
 */
hbool_t H5FD_dsm_steering_is_enabled(const char *name);

/* Description:
 * Pause and wait until completion of steering orders, released by a
 * play.
 */
herr_t H5FD_dsm_steering_wait();

/* Description:
 * Begin/End query - Avoid to open and request file lock acquisition
 * multiple times.
 */
herr_t H5FD_dsm_steering_begin_query();
herr_t H5FD_dsm_steering_end_query();

/* Description:
 * Get/Free DSM handle to interaction dataset - can be passed
 * to HDF5 common functions for further read/write.
 */
herr_t H5FD_dsm_steering_get_handle(const char *name, hid_t *handle);
herr_t H5FD_dsm_steering_free_handle(hid_t handle);

/* Description:
 * Return true if the object exists in the "Interactions" group.
 */
herr_t H5FD_dsm_steering_is_set(const char *name, hbool_t *set);

/* Description:
 * Get/Set the scalar value corresponding to the property name
 * given in the template.
 */
herr_t H5FD_dsm_steering_scalar_get(const char *name, hid_t mem_type,
  void *data);
herr_t H5FD_dsm_steering_scalar_set(const char *name, hid_t mem_type,
  void *data);

/* Description:
 * Get/Set the vector values corresponding to the property name
 * given in the template.
 */
herr_t H5FD_dsm_steering_vector_get(const char *name, hid_t mem_type,
  hsize_t number_of_elements, void *data);
herr_t H5FD_dsm_steering_vector_set(const char *name, hid_t mem_type,
  hsize_t number_of_elements, void *data);
```

Annex D.

Application Integration

In this annex, we show examples of integration of the ICARUS framework into the simulation codes presented in chapter 5. For clarity, note that not all the functionality and integration development is presented below but only snippets of code that are representative of most common tasks one would need to perform.

D.1. Fortran Code Example: SPH-flow

```
PROGRAM SPHFLOW
  use H5
#ifdef USE_H5FD_DSM
  use h5fddsm ! Required module for "dsm" driver
  use h5fddsm_steering ! Required module for steering interface
#endif

  CALL MPI_Init(ierr)
  ...
  CALL H5open_f(error)
  ...
#ifdef USE_H5FD_DSM
  ! Initialize the steering interface
  if (enable_DSM) call h5fd_dsm_steering_init_f(MPI_COMM_WORLD, error)
#endif
  ...
  call setupFromDataFile(h5_filename)
  ...
  ! Launch of calculation
  IF (myrank==0) write(*,*) '  !Launch of calculation', myrank
  ...
  ! Main loop in time
```

```
  do while( isInitial .or. &
    ((numerical_or_physical_dt=='physical')  .and. (t < tmax)).or. &
    ((numerical_or_physical_dt=='numerical') .and. (n_time_step < nbdt)) )
    call Read_H5(h5_filename, &
                 isInitial, &   ! it's the initial file to be loaded
                 fsi_enabled, & ! the user uses Aster on some bodies
                 dsm_ReloadBody)! The user has modified a body via the DSM

#ifdef USE_H5FD_DSM
    if (enable_DSM) then
      ! at the first step, write the data to visualize them in-situ and
      ! initialize the time range
      if(n_time_step == 0) then
        save_step = .true.
        if (myrank == 0) write(*,*) "write to DSM"
        ...
        call h5fd_dsm_steering_vector_set_f("TimeRangeInfo",
          H5T_NATIVE_DOUBLE, numelem, TimeRange, error)
        ...
        call Output_Writing(Neighbour_Search_Required,n_time_step)
        if (myrank == 0) write(*,*) "end write to DSM"
      endif

      ! If waitForGui then make the code wait for a resume command
      if (waitForGui > 0) then
        call h5fd_dsm_steering_wait_f(error)
      endif
      ! cache handles before making queries (avoid excess mpi traffic)
      call h5fd_dsm_steering_begin_query_f(error)
      if (myrank == 0) then
        ! Only check for WaitForGui in the DSM on rank 0 to avoid
        ! excess traffic to the DSM
        call h5fd_dsm_steering_is_set_f("WaitForGui", modified, error)
        if (modified > 0) then
          call h5fd_dsm_steering_scalar_get_f("WaitForGui", &
            H5T_NATIVE_INTEGER, waitForGui, error)
        endif
      endif
      ! Then broadcast the result to the other processes
      call MPI_Bcast(waitForGui, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, error)
      ! Make use of DSM steering interface for other parameters
      call steer_reloadInletVelocity(InletVelocity)
      call steer_reloadBody(dsm_ReloadBody)
      call steer_reloadForce(SteeringForce)
      call steer_reloadMomentum(SteeringMomentum)
```

```
68        call steer_reloadWaveMaker(amplitude,frequence)
69        ! Close cache handle
70        call h5fd_dsm_steering_end_query_f(error)
71
72        ! recall read_h5 to reload the body if the user has modified it
73        ! via the GUI, make use of HDF5 API for reading data
74        if(dsm_ReloadBody) call Read_H5(h5_filename, &
75                    isInitial, &   ! it's the initial file to be loaded
76                    fsi_enabled, & ! the user uses Aster on some bodies
77                    dsm_ReloadBody)! The user has modified a body via the DSM
78      endif
79  #endif
80
81      istep=floor(t/(dt))
82      PHASE_CREATE_DYNAMIC(tTimeStep, trim(line))
83      PHASE_START(tTimeStep)
84
85      CALL Remove_Wrong_Particles(has_removed_IO)
86      Ns_required = Ns_required .or. has_removed_IO
87      ...
88      CALL Time_Step
89
90      is_verlet_exceeded = Verlet_Condition(Verlet_distance, &
91        Verlet_Ntimestep)
92      Ns_required =  Ns_required .OR. is_verlet_exceeded
93
94      PHASE_CREATE_STATIC(tLoadBalance, 'Load Balance/Interactions')
95      PHASE_START(tLoadBalance)
96
97      CALL Load_Balance(has_load_balanced)
98      Ns_required = Ns_required .or. has_load_balanced
99
100     CALL MPI_Allreduce(Ns_required, Neighbour_Search_Required, 1, &
101       MPI_LOGICAL, MPI_LOR, MPI_COMM_WORLD, ier)
102     IF ((Choice_Time_Integration_Solver==4) .OR. &
103         (Choice_Time_Integration_Solver==5)) THEN
104       Neighbour_Search_Required = .TRUE.
105     ENDIF
106     IF( Neighbour_Search_Required ) THEN
107       CALL Interaction_Lists
108       ...
109     ELSE
110       Verlet_Ntimestep = Verlet_Ntimestep + 1
111     ENDIF ! Neighbour_Search_Required
112     PHASE_STOP(tLoadBalance)
```

```
113
114     iappel=1
115     SELECT CASE (Choice_Time_Integration_Solver)
116     CASE(1)
117       PHASE_CREATE_STATIC(tTimeIntegration, &
118         'Time Integration with forward Euler')
119       PHASE_START(tTimeIntegration)
120
121       CALL Explicit_Euler(Neighbour_Search_Required)
122     CASE(2)
123       PHASE_CREATE_STATIC(tTimeIntegration, 'Time Integration with RK4')
124       PHASE_START(tTimeIntegration)
125
126       CALL RungeKutta4(Neighbour_Search_Required)
127     CASE(3)
128       PHASE_CREATE_STATIC(tTimeIntegration, 'Time Integration with SSP43')
129       PHASE_START(tTimeIntegration)
130
131       CALL SSP43(Neighbour_Search_Required)
132     CASE(4)
133       PHASE_CREATE_STATIC(tTimeIntegration, &
134         'Time Integration with explicit incompressible')
135       PHASE_START(tTimeIntegration)
136
137       CALL Explicit_Incompressible(Neighbour_Search_Required)
138     CASE(5)
139       CALL Semi_Implicit_Incompressible(Neighbour_Search_Required)
140     CASE(6)
141       CALL SSPRK83(Neighbour_Search_Required)
142     END SELECT
143     PHASE_STOP(tTimeIntegration)
144
145     t=t+dtc
146
147 #ifdef USE_H5FD_DSM
148     if (enable_DSM) then
149       if (dsm_ReloadBody) save_step = .true.
150       dsm_ReloadBody = .false.
151     endif
152 #endif
153     call OutputStatistics
154     ! Output_Writing is modified to make use of h5pset_fapl_dsm_f
155     call Output_Writing(Neighbour_Search_Required,n_time_step)
156
157     PHASE_STOP(tTimeStep)
```

```
158     n_time_step=n_time_step+1
159   enddo !WHILE( t<tmax )
160
161   ! End of calculation
162   CALL deallocations
163
164   ! Calling H5close automatically closes the DSM connection
165   CALL H5close_f(error)
166   call MPI_Finalize(ierr)
167
168 END PROGRAM SPHFLOW
```

D.2. C++ Code Example: SPH-ALE Code

```
1  #include <hdf5.h>
2  #include <H5FDdsm.h> // Required for "dsm" driver
3  #include <H5FDdsmSteering.h> // Required for steering interface
4
5  int
6  main(int argc, char *argv[])
7  {
8    // Check parameters
9    ...
10   //Create a GPU manager object and retrieve the best available device
11   GpuManager gpuManager;
12   ...
13   // Initialize MPI
14   if ((mpiRv = MPI_Init(&argc, &argv)) != MPI_SUCCESS)
15   {
16     fprintf(stderr, "failed.\n");
17     error("Failed to initialize MPI");
18     goto ExitFunc;
19   }
20   //Set the minimum number of particules per processor
21   parallelObject.setMinimunNumberOfParticulesPerProcessor(10000);
22   // CREATE PARTICLE REPOSITORY THAT WILL BE USED ONLY FOR INITIAL
23   // DOMAIN DECOMPOSITION
24   initPartRepos = new ParticleRepository<opt_tp_dim, opt_tp_order,
25       opt_tp_multiphase, opt_tp_surfaceTension, opt_tp_riemannSolver,
26       opt_tp_riemannSolverMultiphase>
27   (&parallelObject, PARTICLE_MODE_INIT);
28   // CREATE PARTICLE REPOSITORY TO STORE ALL PARTICLES
29   partRepos = new ParticleRepository<opt_tp_dim, opt_tp_order,
30       opt_tp_multiphase, opt_tp_surfaceTension, opt_tp_riemannSolver,
```

```
      opt_tp_riemannSolverMultiphase>
  (&parallelObject, PARTICLE_MODE_REAL);

  // Read configuration
  fprintf(stderr, "Read configuration...'%s' \n", argv[1]);
  ...
  /*******************************************/
  /* INITIAL PARALLEL DOMAIN DECOMPOSITION   */
  /*******************************************/
  ...

RunSolver:

  // Run the solver
  iterativSolver<opt_tp_dim,opt_tp_order,opt_tp_multiphase,
  opt_tp_symmetricInteraction, opt_tp_kernelFunction,
  opt_tp_reconstruction, opt_tp_riemannSolver,
  opt_tp_riemannSolverMultiphase, opt_tp_limiter, opt_tp_correctBound,
  opt_tp_motion, opt_tp_surfaceTension, opt_tp_outputLevel>
  (partRepos,
      config,
      &parallelObject,
      parallelTopology,
      givenPhysicalTime,
      givenInjectionTime,
      config->inputConfig.isSuite,
      givenInjectedMass,
      givenInjectedVolume,
      givenOutputTime,
      asphodeleSHM,
      savedTimeStepValues,
      savedStepCount,
      &config->symetryPlan,
      &ts_partRepos_0,
      &ts_partRepos_1,
      &ts_partRepos_2,
      &gpuManager);

ExitFunc:
  if (initPartRepos) {
    delete initPartRepos;
    initPartRepos = NULL;
  }
  if (partRepos) {
    delete partRepos;
```

```
76      partRepos = NULL;
77    }
78    if (mpiStarted) MPI_Finalize();
79    return rv;
80  }
81
82  template<TP_Dimension tp_dim, TP_Order tp_order,
83  TP_Multiphase tp_multiphase,
84  TP_SymmetricInteractions tp_sym, TP_KernelFunction tp_kernel,
85  TP_Reconstruction tp_reconstruction, TP_RiemannSolver tp_riemann,
86  TP_RiemannSolverMultiphase tp_riemannMulti,TP_Limiter tp_limiter,
87  TP_correctionBoundary tp_correctBound, TP_Motion tp_motion,
88  TP_SurfaceTension tp_surfaceTension, TP_OutputLevel tp_outputLevel>
89  void iterativSolver(ParticleRepository<tp_dim, tp_order, tp_multiphase,
90      tp_surfaceTension, tp_riemann, tp_riemannMulti> *p_partRepos,
91      AsphodeleConfig *p_config,
92      ParallelObject *p_parallelObject,  // OK FROM MPI
93      ParallelTopology *p_parallelTopology,
94      asphodouble p_givenPhysicalTime,
95      asphodouble p_givenInjectionTime,
96      bool p_restartComputation,
97      asphodouble p_givenInjectedMass,
98      asphodouble p_givenInjectedVolume,
99      asphodouble p_givenOutputTime,
100     AsphodeleSHM *p_asphodeleSHM,
101     asphodouble *p_savedTimeStepValues,
102     int p_savedStepCount,
103     SymetryPlan *p_symetryPlan,
104     TimeStageParticleRepository<tp_dim, tp_order, tp_multiphase,
105     tp_surfaceTension, tp_riemann, tp_riemannMulti> *p_ts_partRepos_0,
106     TimeStageParticleRepository<tp_dim, tp_order, tp_multiphase,
107     tp_surfaceTension, tp_riemann, tp_riemannMulti> *p_ts_partRepos_1,
108     TimeStageParticleRepository<tp_dim, tp_order, tp_multiphase,
109     tp_surfaceTension, tp_riemann, tp_riemannMulti> *p_ts_partRepos_2,
110     GpuManager* gpumanager)
111 {
112 #ifdef USE_H5FD_DSM
113   // Initialize the steering interface
114   H5FD_dsm_steering_init(MPI_COMM_WORLD);
115 #endif
116   switch(p_config->temporalIntegration)
117   {
118     case EXPLICIT_EULER:
119       break;
120     case HEUN :
```

```
121       ...
122       break;
123
124     case RK3 :
125       ...
126       break;
127     case RK4 :
128       ...
129       break;
130     default:
131       exit(1);
132   }
133   ...
134   if (p_restartComputation)
135     physicalTime = p_givenPhysicalTime;
136   else
137     physicalTime = 0.;
138
139   iterationCount = 0;
140
141 BeginLoop:
142   fprintf(stderr, "Begin iterative loop...\n");
143   /*****************************/
144   /* Start of iterative loop  */
145   /*****************************/
146   endSimulation = false;
147
148   while(!endSimulation) {
149 #ifdef USE_H5FD_DSM
150     // Get user commands (e.g., pause, resume)
151     H5FD_dsm_steering_update();
152     // Check for jet diameter parameter
153     hbool_t isNewJetDiameterSet = 0;
154     H5FD_dsm_steering_is_set("NewJetDiameter", &isNewJetDiameterSet);
155     printf("New Jet diameter: %lf\n", this->NewJetDiameter);
156     // If the parameter is found in the DSM, pick up the new value
157     if (isNewJetDiameterSet) {
158       H5FD_dsm_steering_scalar_get("NewJetDiameter", H5T_NATIVE_DOUBLE,
159           &this->NewJetDiameter);
160       printf("New Jet Diameter modified to: %lf\n", this->NewJetDiameter);
161       for (int i = 0; i < p_config->jetCount; i++)
162         p_config->jet[i].radius =
163             this->NewJetDiameter * this->OriginalJetDiameter;
164     }
165     // steering for changeRunnerVelocity
```

```
166     // steering for moveSolid
167     // steering for turnDeflector
168 #endif
169
170     /*****************************************************************/
171     /* Begin part specific to computations                           */
172     /*****************************************************************/
173     ...
174     EndIteration:
175     /*****************************************************************/
176     /* End part specific to computations                             */
177     /*****************************************************************/
178
179     // For HDF use all the processes for write operation
180     if (p_config->outputConfig.type == CONFIG_HDF5_OUTPUT) {
181       MPI_Allreduce(&doOutput,
182           &globaldoOutput,
183           1,
184           MPI_INT,
185           MPI_MAX,
186           MPI_COMM_WORLD);
187     } else {
188       globaldoOutput = doOutput;
189     }
190     if (globaldoOutput) {
191       startTimer("Write results");
192       // Write XDMF description template
193       // In this case, the simulation generates the description template
194       // automatically so that HDF5 files can be visualized in-situ
195       if(saveStepIndex == p_savedStepCount &&
196           (p_parallelObject->getMainProcessorIndex() == 0)) {
197         writeLxmfFile(tp_order,
198             tp_dim,
199             tp_multiphase,
200             tp_surfaceTension,
201             tp_outputLevel,
202             tp_riemann,
203             tp_riemannMulti,
204             p_config->solidCount,
205             p_config->jetCount,
206             p_partRepos->virtualFluidParticleArray->getSize() != 0,
207             p_partRepos->virtualWallParticleArray->getSize() != 0,
208             p_partRepos->virtualInletParticleArray->getSize() != 0);
209       }
210
```

```
211       // Data is written using HDF5, hence make use of H5Pset_fapl_dsm
212       writeAsphodeleResults<opt_tp_dim, opt_tp_order, opt_tp_multiphase,
213       tp_surfaceTension, tp_riemann, tp_riemannMulti, tp_outputLevel>
214       (p_config->outputConfig.type,
215           p_partRepos,
216           p_savedTimeStepValues,
217           &saveStepIndex,
218           physicalTime,
219           p_config->solidCount,
220           p_config->solid,
221           physicalTime,
222           p_config->jet,
223           p_config->jetCount,
224           p_parallelTopology);
225
226       // Set the flag back to 0
227       doOutput = 0;
228       globaldoOutput = 0;
229     }
230     ...
231   }
232   /*************************/
233   /* End of iterative loop */
234   /*************************/
235   ...
236
237 ExitFunc:
238   // Free pointers
239   // Frees the grid memory
240 }
```

Bibliography

[1] Partitioned Global Adress Space (PGAS). Available from: http://www.pgas.org.

[2] Message Passing Interface Forum (1997). MPI-2: Extensions to the message-passing interface. Available from: http://www.mpi-forum.org/docs/docs.html.

[3] Adelmann, A., Gsell, A., Oswald, B., Schietinger, T., Bethel, W., Shalf, J., Siegerist, C., and Stockinger, K. Progress on H5Part: A Portable High Performance Parallel Data Interface for Electromagnetics Simulations. In *Particle Accelerator Conference, 2007. PAC. IEEE*, pp. 3396–3398 (2007). doi:10.1109/PAC.2007.4440437.

[4] Ahrens, J., Brislawn, K., Martin, K., Geveci, B., Law, C. C., and Papka, M. Large-Scale Data Visualization Using Parallel Data Streaming. *IEEE Computer Graphics and Applications*, **21** (2001), 34. doi:10.1109/38.933522.

[5] Allen, G., Benger, W., Goodale, T., Hege, H.-C., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J. The Cactus Code: A Problem Solving Environment for the Grid. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing*, HPDC, pp. 253–260. IEEE (2000). ISBN 0-7695-0783-2. doi:10.1109/HPDC.2000.868657.

[6] Alverson, R., Roweth, D., and Kaplan, L. The Gemini System Interconnect. In *IEEE 18th Annual Symposium on High-Performance Interconnects*, HOTI, pp. 83–87 (2010). ISBN 978-1-4244-8547-5. doi:10.1109/HOTI.2010.23.

[7] Argonne National Laboratory. MPICH2. Available from: http://www.mcs.anl.gov/research/projects/mpich2.

[8] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. Toward a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth International Symposium on High Performance Distributed Computing*, pp. 115–124 (1999). doi:10.1109/HPDC.1999.805289.

[9] Biddiscombe, J., Graham, D., and Maruzewski, P. Visualization and analysis of SPH data. *ERCOFTAC Bulletin*, **76** (2008), 9. Available from: http://infoscience.epfl.ch/record/125581/files/InteractiveVisualizationSPH-Spheric-2007-bid-gra-mar.pdf.

[10] Bonachea, D. and Duell, J. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, **1** (2004), 91. doi:10.1504/IJHPCN.2004.007569.

[11] Brightwell, R., Predretti, K., Underwood, K., and Hudson, T. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *Micro, IEEE*, **26** (2006), 41. doi:10.1109/MM.2006.65.

[12] Brooke, J., Coveney, P., Harting, J., Jha, S., Pickles, S., Pinning, R., and Porter, A. Computational Steering in RealityGrid. In *Proceedings of UK e-Science All Hands Meeting 2003*, pp. 885–888 (2003). Available from: http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/179.pdf.

[13] Bruggencate, M. and Roweth, D. DMAPP—An API for One-sided Program Models on Baker Systems. In *Proceedings of Cray User Group* (2010).

[14] Brunner, J. D., Jablonowski, D. J., Bliss, B., and Haber, R. B. VASE: the visualization and application steering environment. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, SC, pp. 560–569. ACM (1993). ISBN 0-8186-4340-4. doi:10.1145/169627.169799.

[15] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J., and Favre, J. Remote Large Data Visualization in the ParaView Framework. In *Eurographics Symposium on Parallel Graphics and Visualization* (edited by B. Raffin, A. Heirich, and L. P. Santos), pp. 163–170. Eurographics Association (2006). ISBN 3-905673-40-1. doi:10.2312/EGPGV/EGPGV06/163-170.

[16] Chaarawi, M., Gabriel, E., Keller, R., Graham, R., Bosilca, G., and Dongarra, J. OMPIO: A Modular Software Architecture for MPI I/O. In *Recent Advances in the Message Passing Interface* (edited by Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra), vol. 6960 of *Lecture Notes in Computer Science*, pp. 81–89. Springer Berlin / Heidelberg (2011). ISBN 978-3-642-24448-3. doi:10.1007/978-3-642-24449-0_11.

[17] Cheriton, D. R. Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems. *SIGOPS Operating Systems Review*, **19** (1985), 26. doi:10.1145/858336.858338.

[18] Childs, H. Architectural Challenges and Solutions for Petascale Postprocessing. *Journal of Physics: Conference Series*, **78** (2007), 012012. doi:10.1088/1742-6596/78/1/012012.

[19] Childs, H., Brugger, E., Bonnell, K., Meredith, J., Miller, M., Whitlock, B., and Max, N. A Contract Based System For Large Data Visualization. In *Visualization, 2005. VIS 05. IEEE*, pp. 191–198 (2005). doi:10.1109/VISUAL.2005.1532795.

[20] Clarke, J. A. Emulating Shared Memory to Simplify Distributed-Memory Programming. *Computational Science Engineering, IEEE*, **4** (1997), 55. doi:10.1109/99.590858.

[21] Clarke, J. A. and Mark, E. R. Enhancements to the eXtensible Data Model and Format (XDMF). In *DoD High Performance Computing Modernization Program Users Group Conference*, HPCMP-UGC, pp. 322–327 (2007). ISBN 0-7695-3088-5. doi:10.1109/HPCMP-UGC.2007.30.

[22] Clarke, J. A. and Namburu, R. R. A distributed computing environment for interdisciplinary applications. *Concurrency and Computation: Practice and Experience*, **14** (2002), 1161. doi:10.1002/cpe.685.

[23] Clarke, J. A. and Namburu, R. R. A Generalized Method for One-Way Coupling of CTH and Lagrangian Finite-Element Codes With Complex Structures Using the Interdisciplinary Computing Environment. Tech. rep., US Army Research Laboratory, Aberdeen Proving Ground, Md. (2004). ARL-TN-230.

[24] Community Research and Development Information Service (CORDIS). NextMuSE—Next generation Multi-mechanics Simulation Environment (2009–2012). Available from: http://cordis.europa.eu/fp7/ict/fet-open/portfolio-nextmuse_en.html.

[25] Cray. SHMEM technical note for C. Tech. rep., Cray Research, Inc. (1994). SG-2516 2.3.

[26] Docan, C., Parashar, M., and Klasky, S. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC, pp. 25–36. ACM (2010). ISBN 978-1-60558-942-8. doi:10.1145/1851476.1851481.

[27] Ducrozet, G., Bonnefoy, F., Touzé, D. L., and Ferrant, P. A modified High-Order Spectral method for wavemaker modeling in a numerical wave tank. *European Journal of Mechanics - B/Fluids*, **34** (2012), 19. doi:10.1016/j.euromechflu.2012.01.017.

[28] Fabian, N., Moreland, K., Thompson, D., Bauer, A. C., Marion, P., Geveci, B., Rasquin, M., and Jansen, K. E. The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library. In *IEEE Symposium on Large-Scale Data Analysis and Visualization*, LDAV (2011).

[29] Fujitsu. K computer (2011). Available from: `http://www.fujitsu.com/global/about/tech/k`.

[30] Gomez-Gesteira, M., Rogers, B. D., Dalrymple, R. A., and Crespo, A. J. C. State-of-the-art of classical SPH for free-surface flows. *Journal of Hydraulic Research*, **48** (2010), 6. `doi:10.1080/00221686.2010.9641242`.

[31] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. Interactive SPH Simulation and Rendering on the GPU. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation* (edited by Z. Popovic and M. Otaduy), pp. 55–64. Eurographics Association (2010). ISBN 978-3-905674-27-9. `doi:10.2312/SCA/SCA10/055-064`.

[32] Graham, R. The MPI 2.2 Standard and the Emerging MPI 3 Standard. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (edited by M. Ropo, J. Westerholm, and J. Dongarra), vol. 5759 of *Lecture Notes in Computer Science*, pp. 2–2. Springer Berlin / Heidelberg (2009). ISBN 978-3-642-03769-6. `doi:10.1007/978-3-642-03770-2_2`.

[33] Gropp, W., Lusk, E., and Thakur, R. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA (1999). ISBN 0-262-57132-3.

[34] Gropp, W. and Thakur, R. Revealing the Performance of MPI RMA Implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (edited by F. Cappello, T. Herault, and J. Dongarra), vol. 4757 of *Lecture Notes in Computer Science*, pp. 272–280. Springer Berlin / Heidelberg (2007). ISBN 978-3-540-75415-2. `doi:10.1007/978-3-540-75416-9_38`.

[35] Harris, M. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH. ACM (2005). `doi:10.1145/1198555.1198768`.

[36] Hilbrich, T., Müller, M., Schulz, M., and de Supinski, B. Order Preserving Event Aggregation in TBONs. In *Recent Advances in the Message Passing Interface* (edited by Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra), vol. 6960 of *Lecture Notes in Computer Science*, pp. 19–28. Springer Berlin / Heidelberg (2011). ISBN 978-3-642-24448-3. `doi:10.1007/978-3-642-24449-0_5`.

[37] Howison, M., Koziol, Q., Knaak, D., Mainzer, J., and Shalf, J. Tuning HDF5 for Lustre File Systems. In *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage* (2010). LBNL-4803E. Available from: http://www.hdfgroup.org/pubs/papers/howison_hdf5_lustre_iasds2010.pdf.

[38] IBM. General Parallel File System (GPFS). Available from: http://www.ibm.com/systems/software/gpfs.

[39] Kitware Inc. *The VTK User's Guide*. Kitware Inc. (2010). ISBN 978-1-930934-23-8.

[40] Kohl, J. A., Wilde, T., and Bernholdt, D. E. CUMULVS: Interacting with High-Performance Scientific Simulations, for Visualization, Steering and Fault Tolerance. *Int. J. High Perform. Comput. Appl.*, **20** (2006), 255. doi:10.1177/1094342006064502.

[41] Lawrence Livermore National Laboratory. ASC Sequoia Request for Proposals (2008). Available from: https://asc.llnl.gov/sequoia/rfp.

[42] Li, J., et al. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC. ACM (2003). ISBN 1-58113-695-1. doi:10.1145/1048935.1050189.

[43] Linsen, L., Hamann, B., Joy, K., Pascucci, V., and Duchaineau, M. Wavelet-Based Multiresolution with $\sqrt[n]{2}$ Subdivision. *Computing*, **72** (2004), 129. doi:10.1007/s00607-003-0052-0.

[44] Lofstead, J., Zheng, F., Klasky, S., and Schwan, K. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *IEEE International Symposium on Parallel and Distributed Processing*, IPDPS, pp. 1–10 (2009). doi:10.1109/IPDPS.2009.5161052.

[45] Lorensen, W. E. and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH, pp. 163–169. ACM (1987). ISBN 0-89791-227-6. doi:10.1145/37401.37422.

[46] Lorenz, D., Buchholz, P., Uebing, C., Walkowiak, W., and Wismüller, R. Steering of sequential jobs with a distributed shared memory based model for online steering. *Future Gener. Comput. Syst.*, **26** (2010), 155. doi:10.1016/j.future.2009.05.016.

[47] Los Alamos National Laboratory. Roadrunner. Available from: http://www.lanl.gov/roadrunner.

[48] Malakar, P., Natarajan, V., and Vadhiyar, S. S. InSt: An Integrated Steering Framework for Critical Weather Applications. *Procedia Computer Science*, **4** (2011), 116. Proceedings of the International Conference on Computational Science, ICCS. doi:10.1016/j.procs.2011.04.013.

[49] Marongiu, J.-C., Leboeuf, F., Caro, J., and Parkinson, E. Free surface flows simulations in Pelton turbines using an hybrid SPH-ALE method. *Journal of Hydraulic Research*, **48** (2010), 40. doi:10.3826/jhr.2010.0002.

[50] McCormick, B. H., DeFanti, T. A., and Brown, M. D. Visualization in Scientific Computing. *Computer Graphics*, **21** (1987). doi:10.1145/43965.43966.

[51] Monaghan, J. An introduction to SPH. *Computer Physics Communications*, **48** (1988), 89. doi:10.1016/0010-4655(88)90026-4.

[52] Moreland, K., et al. Examples of In Transit Visualization. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, PDAC. IEEE/ACM (2011).

[53] Mulder, J. D., van Wijk, J. J., and van Liere, R. A survey of computational steering environments. *Future Generation Computer Systems*, **15** (1999), 119. doi:10.1016/S0167-739X(98)00047-8.

[54] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, **20** (2006), 203. doi:10.1177/1094342006064503.

[55] Oak Ridge Leadership Computing Facility. Jaguar. Available from: http://www.olcf.ornl.gov/computing-resources/jaguar.

[56] Oak Ridge Leadership Computing Facility. Titan. Available from: http://www.olcf.ornl.gov/computing-resources/titan.

[57] Object Management Group. Common Object Request Broker Architecture (CORBA). Available from: http://www.omg.org/spec/CORBA.

[58] Oger, G., Jacquin, E., Guilcher, P.-M., Brosset, L., Deuff, J.-B., Le Touzé, D., and Alessandrini, B. Simulations of complex hydro-elastic problems using the parallel SPH model SPH-Flow. In *Proceedings of the 4th SPHERIC* (2009).

[59] Oracle. Lustre. Available from: http://www.lustre.org.

[60] Parker, S. G. and Johnson, C. R. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, SC. ACM (1995). ISBN 0-89791-816-9. doi:10.1145/224170.224354.

[61] Parker, S. G., Weinstein, D. W., and Johnson, C. R. Modern Software Tools for Scientific Computing. chap. The SCIRun Computational Steering Software System, pp. 5–44. Birkhauser Boston Inc., Cambridge, MA, USA (1997). ISBN 0-8176-3974-8.

[62] Pritchard, H., Gorodetsky, I., and Buntinas, D. A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE. In *Recent Advances in the Message Passing Interface* (edited by Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra), vol. 6960 of *Lecture Notes in Computer Science*, pp. 110–119. Springer Berlin / Heidelberg (2011). ISBN 978-3-642-24448-3. doi:10.1007/978-3-642-24449-0_14.

[63] Rew, R. and Davis, G. NetCDF: An Interface for Scientific Data Access. *Computer Graphics and Applications, IEEE*, **10** (1990), 76. doi:10.1109/38.56302.

[64] Rew, R., Hartnett, E., and Caron, J. NetCDF-4: Software Implementing an Enhanced Data Model for the Geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*, AMS (2006). Available from: http://ams.confex.com/ams/pdfpapers/104931.pdf.

[65] Richart, N., Esnard, A., and Coulaud, O. Toward a Computational Steering Environment for Legacy Coupled Simulations. In *Sixth International Symposium on Parallel and Distributed Computing*, ISPDC, p. 43 (2007). doi:10.1109/ISPDC.2007.55.

[66] Ross, R. B., Peterka, T., Shen, H.-W., Hong, Y., Ma, K.-L., Yu, H., and Moreland, K. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series*, **125** (2008), 012099. doi:10.1088/1742-6596/125/1/012099.

[67] Sandia Corporation. ASCI Red Web Site (2003). Available from: http://www.sandia.gov/ASCI/Red.

[68] Sandia Corporation. Introducing Red Storm (2007). Available from: http://www.sandia.gov/ASC/redstorm.html.

[69] Silva, C., Chiang, Y., Corrêa, W., El-Sana, J., and Lindstrom, P. Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics. In *Visualization'02 Course Notes* (2002).

[70] Squillacote, A. H. *The ParaView Guide, A Parallel Visualization Application*. Kitware Inc. (2008). ISBN 1-930934-21-1.

[71] Thakur, R., Gropp, W., and Lusk, E. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, **28** (2002), 83. doi:10.1016/S0167-8191(01)00129-6.

[72] The HDF Group. Hierarchical Data Format Version 5. Available from: http://www.hdfgroup.org/HDF5.

[73] The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. Available from: http://mvapich.cse.ohio-state.edu/index.shtml.

[74] Vishwanath, V., Hereld, M., Morozov, V., and Papka, M. E. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pp. 19:1–19:11. ACM (2011). ISBN 978-1-4503-0771-0. doi:10.1145/2063384.2063409.

[75] Walker, D. and Otto, S. Redistribution of Block-Cyclic Data Distributions Using MPI. *Concurrency — Practice and Experience*, **8** (1996), 707. doi:10.1002/(SICI)1096-9128(199611)8:9<707::AID-CPE269>3.0.CO;2-V.

[76] Whitlock, B., Favre, J. M., and Meredith, J. S. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Eurographics Symposium on Parallel Graphics and Visualization* (edited by T. Kuhlen, R. Pajarola, and K. Zhou), pp. 101–109. Eurographics Association (2011). ISBN 978-3-905674-32-3. doi:10.2312/EGPGV/EGPGV11/101-109.

[77] Yu, H., Wang, C., Grout, R., Chen, J., and Ma, K.-L. In Situ Visualization for Large-Scale Combustion Simulations. *Computer Graphics and Applications, IEEE*, **30** (2010), 45. doi:10.1109/MCG.2010.55.

[78] Zhang, F., Docan, C., Parashar, M., and Klasky, S. Enabling Multi-physics Coupled Simulations within the PGAS Programming Framework. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pp. 84–93 (2011). doi:10.1109/CCGrid.2011.73.

[79] Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S. SCIRun2: a CCA framework for high performance computing. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 72–79 (2004). doi:10.1109/HIPS.2004.1299192.

List of Refereed Publications

[80] Biddiscombe, J., Soumagne, J., Oger, G., Guibert, D., and Piccinali, J.-G. Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver. In *Eurographics Symposium on Parallel Graphics and Visualization* (edited by T. Kuhlen, R. Pajarola, and K. Zhou), pp. 91–100. Eurographics Association (2011). ISBN 978-3-905674-32-3. Honourable Mention Award. doi:10.2312/EGPGV/EGPGV11/091-100.

[81] Biddiscombe, J., Soumagne, J., Oger, G., Guibert, D., and Piccinali, J.-G. Parallel Computational Steering for HPC Applications using HDF5 Files in Distributed Shared Memory. *IEEE Transactions on Visualization and Computer Graphics*, **18** (2012), 852. doi:10.1109/TVCG.2012.63.

[82] Soumagne, J. and Biddiscombe, J. Computational Steering and Parallel Online Monitoring Using RMA through the HDF5 DSM Virtual File Driver. *Procedia Computer Science*, **4** (2011), 479. Proceedings of the International Conference on Computational Science, ICCS. doi:10.1016/j.procs.2011.04.050.

[83] Soumagne, J., Biddiscombe, J., and Clarke, J. An HDF5 MPI Virtual File Driver for Parallel In-situ Post-processing. In *Recent Advances in the Message Passing Interface* (edited by R. Keller, E. Gabriel, M. Resch, and J. Dongarra), vol. 6305 of *Lecture Notes in Computer Science*, pp. 62–71. Springer Berlin / Heidelberg (2010). ISBN 978-3-642-15645-8. doi:10.1007/978-3-642-15646-5_7.

[84] Soumagne, J., Biddiscombe, J., and Esnard, A. Data Redistribution using One-sided Transfers to In-memory HDF5 Files. In *Recent Advances in the Message Passing Interface* (edited by Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra), vol. 6960 of *Lecture Notes in Computer Science*, pp. 198–207. Springer Berlin / Heidelberg (2011). ISBN 978-3-642-24448-3. doi:10.1007/978-3-642-24449-0_23.

Index