In-Situ Visualization in Computational Fluid Mechanics using Open-Source tools: Integration of Catalyst into *Code_Saturne*

Ribés Alejandro, Lorendeau Benjamin, Jomier Julien and Fournier Yvan

Abstract The volume of data produced by numerical simulations performed on high performance computers is becoming increasingly large. The visualization of these large post-generated volumes of data is currently a bottleneck for the realization of engineering and physics studies in industrial environments. In this context, Catalyst is a prototype in-situ visualization library developed by Kitware to help reduce the data post-treatment overhead. Additionally, *Code_Saturne* is a Computational Fluid Dynamics code developed by EDF, one of the largest electricity producers in Europe, for its large scale simulations. Both Catalyst and *Code_Saturne* are open-source software. In this chapter we present a case study where Catalyst is coupled with *Code_Saturne*. We evaluate the feasibility and performance of this integration by running several use cases in one of our corporate supercomputers.

1 Introduction

Computational Fluid Dynamics (CFD) is a fundamental step for the study and optimization of electricity production. Indeed, current power plants use water as a mean of convective heat transfer. Consequently, the simulation and visualization of fluid dynamics phenomena is of great importance for the

Ribés Alejandro
EDF R&D, Clamart, France e-mail: alejandro.ribes@edf.fr

Lorendeau Benjamin
EDF R&D, Clamart, France, e-mail: benjamin.lorendeau@gmail.com

Jomier Julien
Kitware, Villeurbanne, France e-mail: julien.jomier@kitware.com

Fournier Yvan
EDF R&D, Chatou, France e-mail: yvan.fournier@edf.fr

energy industry. Electricité de France (EDF), one of the largest electricity producer in Europe, has been developing for the past 15 years an open source CFD code named *Code_Saturne*. *Code_Saturne* performs CFD computations on very large models [15]. EDF owns several supercomputers that regularly run this code in order to perform CFD analysis involving large amounts of data. In this context, the post-processing and visualization steps become critical.

EDF also develops, in collaboration with OpenCascade and the French Center of Atomic Research (CEA), an open-source numerical simulation platform called SALOME [12]. This platform provides generic methods for pre- and post-processing of numerical simulations. SALOME is based on an open architecture made of reusable components such as computer-aided design (CAD), meshing, high performance computing (HPC) execution management, multi-physics coupling, data post-processing and visualization. The visualization module of the SALOME platform is currently based on the open-source post-processing platform ParaView. Furthermore, *Code_Saturne* is often used in conjunction with the SALOME platform.

In the past, studies and improvements in scientific simulation have been mainly focused on the solver, due to being the most cycle-consuming part in the simulation process. Thus, visualization has been traditionally run sequentially on a smaller computer and at the very end of the solver computation. At the time, this was easily explained by the small need for both memory and computation resources in most of the visualization cases. Nevertheless, with the increase of our computational capabilities, we tend to use and generate much more data than what we were used to. Thus, as the scale of CFD simulation problems is getting wider, specific issues are emerging related to input/output efficiency. In particular, data generated during the solver computation and used for the visualization are the source of a worrisome overhead. Even worse, some researchers are starting to spend more time writing and reading data than actually running solvers and visualizations [13]. This new trend compels us to design new input/output (I/O) strategies and consider visualization as a part of our high-performance simulation systems.

For some years, *in-situ* visualisation techniques have been successfully applied in different contexts and mainly by research institutes. In this chapter, we present an overview of the efforts needed to transition a traditional simulation code to an *in-situ* model in an industrial environment. This is the reason why care have been taken constructing uses cases that are representative of our current visualisation problems.

Most fluid dynamic engineers at EDF R&D are currently visualizing lower temporal and spatial resolution versions of their simulations in order to avoid I/O bottlenecks when large quantities of data are involved. We decided to address the subject of co-processing and *in-situ* visualization which has been proved to be an effective solution against the current limitations of this problem [6], [14]. Our aim is to provide EDF engineers with an operational research-oriented tool in a mid-term basis. For this, we chose to evaluate

Catalyst as an industrial tool for performing *in-situ* visualization. Catalyst is a library, developed by Kitware, which implements the co-processing for ParaView by defining the visualization process through the ParaView user interface and exploiting VTK's parallel algorithms for the post-processing of data generated by numerical simulation [9].

In this chapter, we report a study upon the effectiveness and scalability of a prototype implementation of the co-processing in an industrial case based on the coupling of *Code_Saturne* with Catalyst. In section 2 we introduce the motivation of this work. In section 3 we discuss related advances on recent visualization *in-situ* systems. We then introduce, in section 4 *Code_Saturne*, the CFD code developed at EDF R&D. In section 5 we present our integration of Catalyst into *Code_Saturne* and how the system is used by EDF users in the context of fluid dynamic simulations. Section 6 describes our use case and presents results on one of our corporate clusters. Finally, section 7 presents our analysis of the results and describes our ongoing and future work.

2 Motivation

Most numerical simulation engineers at EDF R&D are currently visualizing lower temporal and spatial resolution versions of their simulations, in order to avoid I/O issues and cumbersome visualisation procedures, when large quantities of data are involved. We believe that other industries dealing with large simulations are having the same problem. This is the reason why we decided to leverage the power of co-processing and *in-situ* visualization. Our aim is to provide our research-oriented engineers with an operational tool within two years. Thus, we have evaluated Catalyst as an industrial tool for performing *in-situ* visualization.

First of all, it is important to better describe the scope of our industrial visualisation solutions to understand why *in-situ* processing is needed. In Table 1 we show the results of a simple subjective experiment conducted by one of our engineers. At the end of 2012, she meshed a simple cube at different resolutions and then tried to visualise the results giving a subjective evaluation of how she could work. She used our open-source numerical simulation platform, SALOME, and a standard scientific PC with 8Gb of RAM. Table 1 presents the results of her subjective experiment. The study clearly shows that she started working without an immediate system response for meshes which contain more that 10 millions cells and for 50 million cells the system was not responding at all. At the time that this test was performed, some of our R&D engineers were already running simulations with meshes containing around 200 millions cells and, in June 2014, with meshes reaching 400 millions cells. This implies that copying the simulation results from the simulation cluster to the scientific stations is not practical, first because of the long transfer time and second because, as table 1 shows, the visualization

and post-processing tasks cannot even run. It clearly appears that the visualization and post-processing of large meshes is a serious bottleneck in this industrial context. This motivated the beginning of this work.

Table 1: Subjective characterization of the reaction time of the SALOME platform for different mesh sizes.

MESH SIZE MANIPULATION EXPERIMENT					
Number of cells	10 Thousands	100 Thousands	1 Millions	10 Millions	50 Millions
RAM(%)	<50%	<50%	<50%	100%	Saturated
Reaction time	Immediate	Immediate	2 to 3 seconds	Uncomfortable	Not responding

A first solution to the post-processing bottleneck consists in the installation of parallel visualisation servers that can deal with the large amount of data generated by the numerical simulations. In general, such a system (in our case a ParaView "pvserver") is installed on a visualisation cluster; the system reads the data from disk and performs the visualisation operations in parallel, while streaming images (or small 3D models) to a remote client hosted on a standard PC. EDF R&D owns a visualisation cluster as part of its HPC cluster "Ivanhoe", which will be described later in this chapter. This type of solution implies writing and reading large data in parallel. Even if these operations are performed on a cluster with a fast distributed file system, *in-situ* processing provides much better performances as large resulting datasets are potentially never generated.

In order to get more insight, we can model the whole time taken by simulation and visualisation tasks as an addition of individual operations. For the traditional *a posteriori* visualisation approach:

$$t_{posterior} = T_s + T_w + T_r + T_v \tag{1}$$

where T_s is the simulation time, T_w is the time for writing the data, T_r is the time to read the data (either in parallel or sequentially) and T_v is the time to perform visualization operations and probably write visualisation results (like videos, images or graphs). For the *in-situ* approach:

$$t_{in-situ} = T_s + T_{process} + T_{w-in-situ} + T_{v-in-situ} \tag{2}$$

where T_s is the simulation time (the same as in $t_{posterior}$), $T_{process}$ is the time to perform the visualisation operations *in-situ*, $T_{w-in-situ}$ the time to store the already processed visualisation results and $T_{v-in-situ}$ the time that the engineer takes to visualize the videos or other pre-processed data. Comparing these two formulas we can see that $t_{posterior} >>> t_{in-situ}$ as, in the case of *in-situ*, we skip writing and reading large volumes of data, $T_w + T_r >>> T_{process} + T_{w-in-situ}$; but also the visualisation time is reduced $T_v >>> T_{v-in-situ}$ because, in the *a posteriori* approach, visual-

ising means performing operations on large data while in the *in-situ* approach only lightweight data is involved. In the rest of this chapter these times will be exemplified. For instance, in the top two images of figure 4 one can compare T_w and $T_s + T_w$ for different simulations and the relationship $T_w + T_r >>> T_{process} + T_{w-in-situ}$ becomes clear. These two images demonstrate how quickly I/O times widen, relative to solver times, which is why *in-situ* techniques are needed.

In conclusion, the whole process of "simulation + visualisation" is faster when performed *in-situ*, furthermore the volume of the produced data in much smaller. This is the reason that motivated this work.

3 Related Work

The size of generated data has become an important subject in high performance computing, due to the need of a better I/O efficiency in our computing system. To answer this problem, several visualization systems have been created. We can distinguish two main approaches in recent solutions. The first one is to integrate a specific *in-situ* visualization directly to the simulation code. Such approach proved to be an efficient way to provide co-processing for a given simulation as well as a visualization system as it is the case in the hurricane prediction [7] and earthquake simulation [14] systems. This method has been proven to lead to good performances but is limited to a specific implementation.

The second approach is to provide a general post-processing framework letting the simulation and the visualization code communicate together. EPSN which is a general coupling system, allows for the connection of M simulation nodes to N visualization nodes through a network [8]. This solution is a loosely coupled approach, requiring separate resources and data transfer through the network. This approach presents the advantage of not overloading the nodes used for computation. Thus the visualization code does not interfere with the execution of the simulation. Based on the same approach, a ParaView plug-in named ICARUS [3] has been developed. It differs from EPSN in design by having lower requirements as it only needs the use of a single HDF5 library and file driver extension. Such solutions offer tools for researchers to interact with their simulations by allowing them, not only to monitor their current states but also to modify the parameters of the remaining simulation steps. Those computational steering solutions as well as the RealityGrid project [4] focus on interactivity with simulation whereas our main objective is to provide *in-situ* visualization operations to researchers while minimizing I/O overhead and disk space use.

Both built upon the well known parallel visualization library VTK, the application frameworks VisIt [5] and ParaView [10] both provide through the possibility to co-process simulation data via libsim [16] and Catalyst [9]

respectively. Those *in-situ* solutions are tightly coupled and while they limit potential interactions with the running simulation, they also highly reduce the need of network data transfer. Thus, they contribute to circumventing the inefficiency of high performance computing I/O systems. Those solutions take their benefits from directly accessing the simulation memory to perform visualization tasks by simply asking for a pointer to the available data. One major drawback of this approach is the necessity to provide a coherent data layout to those libraries. Moreover, as this type of solution often gains from computing pre-determined visualization tasks, it is not well suited for results exploration. As building a steering solution for Code_Saturne is out of the scope of this case study, we do not consider these drawbacks as a limitation.

After evaluating the performance solutions offered by ParaView and VTK, we choose Catalyst as our co-processing library for our case study as it answers EDF's visualization needs while focusing on the reduce of data amount use. Further, Kitware recently included services allowing the interactions with the running simulation, the so-called Live Catalyst.

4 *Code_Saturne*: A Computational Fluid Dynamics code

Code_Saturne is a computational fluid dynamics software designed to solve the Navier-Stokes equations in the cases of 2D, 2D axisymmetric or 3D flows. Development started in 1997, with a first release in 2000, and the code has been released as free software under a GPL (General Public Licence) licence since 2007. Its main module is designed for the simulation of flows which may be steady or unsteady, laminar or turbulent, incompressible or potentially dilatable, isothermal or not. Scalars and turbulent fluctuations of scalars can be taken into account. The code includes specific modules, referred to as "specific physical models", for the treatment of atmospheric flows, Lagrangian particle tracking, semi-transparent radiative transfer, gas combustion, pulverised coal combustion, electricity effects (Joule effect and electric arcs) and compressible flows. *Code_Saturne* relies on a finite volume discretisation and allows the use of various mesh types which may be hybrid (containing several kinds of elements) and may have structural non-conformities (hanging nodes). The parallelization is based on standard spatial partitioning with ghost cells that facilitate data passing between adjacent cells lying across the boundaries of disconnected parts using the Message Passing Interface. More technical details are presented in [1] and [2], and many resources are available at http://www.code-saturne.org. *Code_Saturne* is also used as a base for the NEPTUNE_CFD code, specialized in multiphase flows, and which uses a different time stepping scheme, but mostly the same volume discretization scheme.

As *Code_Saturne* is used for industrial cases involving complex flows, with turbulence modeling requiring sufficiently fine resolution, large meshes are

often needed. In 2000, the largest meshes used for actual studies were around 1.5 million cells; today, they have reached up to 400 million cells. More common studies use meshes about 10 times smaller than that. Meshes up to 3.2 billion cells have been tested for a few time steps, to ensure the code's internal mechanisms work well at scale.

Code_Saturne focuses on the solver, and its uses requires external tools for the major part of the meshing and visualisation tasks, though the code itself offers major preprocessing features to make these easier, such as parallel joining of independently-built and read submeshes (whether conforming or not), and user-definable post-processing functions. Many input mesh and visualisation output formats are supported (including the EDF and CEA MED format, and the standardized CGNS format).

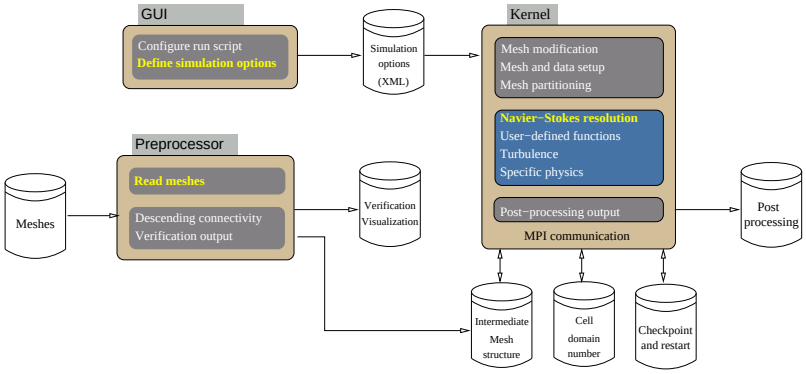


Fig. 1: *Code_Saturne* toolchain components.

The number of separate executable tools is quite reduced, with a few interactive tools and associated commands designed for data setup.

To make the use of HPC as seamless as possible, without multiplying the number of tools or requiring complex libraries or dependencies, mesh, checkpoint/restart, and post-processing output files are partitioned independently: in addition to the connectivity of each local mesh, which is described with a local numbering, global ids are also maintained, for import and export purposes, and for ghost cell to local cell matching. Multiple ranks participate in reading and writing files using MPI-IO.

Typically, computational resource requirements are primarily determined either by the time to solution, or the size of the model. For time to solution, the number of cores may be selected in order to solve the problem in a given time. In practice, optimal performance is often obtained in a range of 30 000 to 60 000 cells per rank on a typical HPC cluster (this being the best compromise between communication latency and cache behavior). On machines with very good network/compute performance ratios, such as IBM Blue Genes or Cray X series, this range may be a bit wider.

5 Using Catalyst

Catalyst is the coprocessing library of ParaView. It has been designed to be tightly coupled with simulation codes to perform in situ analysis at run time. Catalyst leverages the Visualization Toolkit (VTK) for scalable data analysis and visualization. Furthermore, it can be coupled with the ParaView In Situ Analysis framework to perform run-time visualization of data extracts and steering of the data analysis pipeline. Catalyst provides two sets of tools: one for simulation users and one for simulation developers.
For simulation users, it is possible to create a coprocessing pipeline using two different methods. The first method does not require any knowledge of ParaView and relies on pre-generated scripts. These predefined scripts can be written in C++ or Python and are, usually, expected to run without any configuration options. The second method uses the ParaView interface to generate a coprocessing script from scratch and intuitively adjust its parameters as needed. This method is similar to using ParaView interactively to setup desired post-processing pipelines. The goal of these pipelines is to extract post-processed information during the simulation run. Ideally one should start with a representative dataset from the simulation. It is also possible to modify directly the generated Python's scripts which have been previously created using ParaView. However, this would require a knowledge of the ParaView Python application programming interface (API).
For simulation developers, Catalyst provides the tools to create an adaptor between the simulation code and the visualization pipeline. The adaptor binds the simulation code and Catalyst so that both the functions of the simulation code and the general-purpose API of Catalyst can be accessed. As Catalyst itself is independent of the simulation code, only the adaptor has to be developed by the designers of the solver. This flexibility is critical in order to successfully integrate external code into complex simulations usually running with different languages and packages. Catalyst is also easily extensible so that users can deploy new analysis and visualization techniques to existing coprocessing installations. Catalyst provides all the communication and synchronization routine and the pipeline mechanics necessary for coprocessing. Catalyst also provides powerful data processing capabilities through VTK filters as well as many writers and support for compositing and rendering.
The Catalyst library has also been developed with features to address limitations that come with pre-configuring a pipeline, but there may still be some unexpected data in the arbitrary simulation. To address these situations, the pipeline must be adjusted interactively. The Catalyst library can leverage ParaView's client server mechanism to allow an interactive ParaView client to connect to a server running inside an in situ pipeline. ParaView can then read from a Catalyst data source like it reads from a file. This enables construction/modification of a pipeline interactively in the ParaView client via this live data source. Additionally, by enabling the client-server architecture in the Catalyst library, some or all of the data analysis and visualization

pipeline can offload, if desired, to a separate machine, e.g., a smaller visualization cluster with specialized graphics hardware.

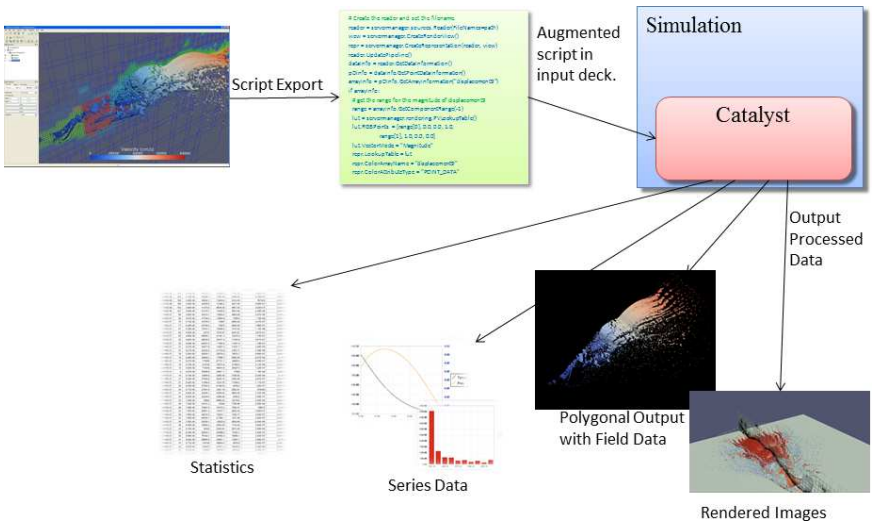


Fig. 2: Overall Catalyst workflow.

5.1 Code_Saturne with Catalyst

Using Catalyst with *Code_Saturne* is quite straightforward, and fits quite naturally in the existing architecture. In this section we describe first how we implemented an adaptor for Catalyst and then we present how we configured the post-processing pipeline.

5.1.1 Adaptor implementation

When embedding Catalyst there is always a non-zero cost in terms of time, memory and number of processors. Naively, one could simply address these issues by requesting a greater number of processors, but in most cases this is not possible nor practical. Therefore, great care must be taken in the implementation of the adaptors. If memory is limited, the adaptor either uses complicated pointer manipulation or uses a smaller region of memory. If memory is not limited, then deep copy of the simulation data structures into a VTK data object doubles the resident memory, and also creates a CPU cost involved with copying the data. The overriding overhead issue in embedding ParaView Catalyst in a simulation code is the memory management in an adaptor translating data structures in the running simulation.

In order to co-process the simulation data, Catalyst must be provided with the data formatted to the VTK data object structure. To accomplish this task, several solutions are possible, depending on the format used for the data of the simulation code. In the case where the format of the simulation code is similar to VTK and, moreover, the simulation data can be shared at any time, then it is possible to feed Catalyst with a direct pointer to the simulation memory. This option is indeed preferred, when possible, as it allows to decrease the memory footprint. Another option is to fully or partially copy the data from the simulation into a VTK object, and then send this object to Catalyst.
As users of *Code_Saturne* are provided with several output formats and as the data structure in our simulation differs from the VTK data object structure, feeding Catalyst with a direct pointer to the simulation memory is not possible. Thus, in this configuration data is copied from the simulation into a VTK data object. In fact, we allocate a vtkDoubleArray data structure to store our data for Catalyst. Furthermore, we provide a pointer of this VTK data structure to *Code_Saturne* so it can transform its simulation data and then fill the VTK data object.
The memory cost increase of our solution can be alleviated by using more machines. The CPU cost of the copy is in a range similar to the one needed when adapting simulation data to a specific output format. This cost is largely affordable comparatively to the time to write data to disk when storing time step specific outputs.

5.1.2 Pipeline configuration

From the point of view of an engineer performing a fluid mechanics simulation using *Code_Saturne*, the workflow of a co-processing simulation is 1) to define a ParaView pipeline describing what the user wants to study and 2) to run the simulation. Since users are already familiar with fluid mechanics simulations, defining the pipeline for the co-processing remains the main sticking point. Thus this new process should be done in an efficient way and should not become a cumbersome bottleneck. This point is of great importance, especially in an industrial environment like ours.
As we have explained in the previous section, the definition of a Catalyst pipeline can be achieve either programmatically or via the ParaView interface. In our industrial context, the former was considered too complicated and time consuming for the end user, especially when setting camera parameters is needed, as no visualization feedback is provided. Therefore, the Catalyst pipeline has been created using the ParaView user's interface. This solution appears to be much easier as one can interact with ParaView in the same way he/she uses to when visualizing the results a posteriori. This solution is also easier to deploy with ParaView using its companion co-processing plugin.

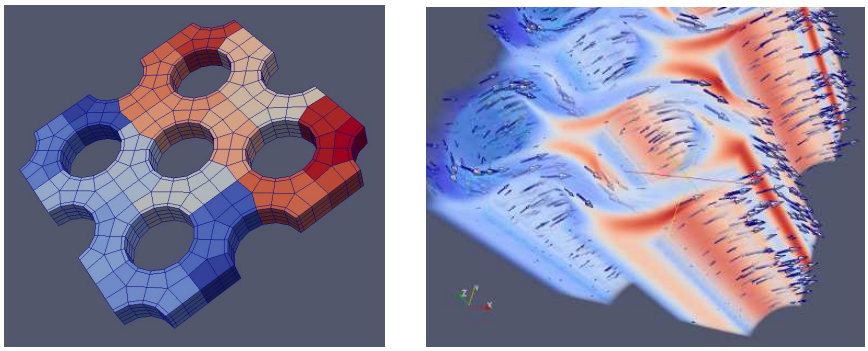
Indeed, using ParaView to define the co-processing requires a representative input dataset. One could use the available resources on a large cluster in order to setup the pipeline. However we chose to provide a simplified or under-sampled version of the large geometry to define the pipeline. In fact, this strategy is possible in ParaView but some characteristics of the initial geometry must be present in its simplified version; more importantly the name of the data fields must remain the same, as they are part of the definition of the pipeline.
The generation of the co-processing pipeline implies several steps. First of all, the users start with a Computer Aided Design (CAD) version of the geometry which is parametrized. This parametric representation can generate meshes at different resolution levels. In our case, this is performed inside the open-source SALOME [12] platform for numerical simulation.
We then generate two different meshes, one at high resolution (up to 204M hexahedrals in the current use cases) that will be used for the CFD simulation and one with a lower resolution to define the pipeline (700 000 hexahedrals in our use cases). The lowest resolution mesh is fed into *Code_Saturne* to perform a short simulation. This allows ParaView to obtain a representation containing not only the geometry but also the result fields. This is the data that is then used to define the pipeline. The different processing pipelines are presented next.

6 Results

6.1 Required User Interactions for Co-processing

Before presenting our results we briefly describe how the user interactions were performed. The following steps were necessary in order to use the developed co-processing technology:
1) A "light version" of the input mesh is generated as explained in section 5.1.2. As the user possesses a CAD version of the geometry that is parametrized, it is then possible to obtain meshes at different spatial resolutions. A "light mesh" of small size in memory and representative of the CAD geometry is obtained. Figure 3a represents the "light version" of the mesh used in our experiments.
2) A short simulation (normally just a few seconds on a local machine) on the "light mesh" is run. This simulation allows to define the information about the result fields needed to create a visualisation pipeline in ParaView (e.g. temperature, pressure, velocity). An "augmented light mesh" is therefore created.
3) The mesh and the fields obtained at the end of step 2 are then read in ParaView and the user defines her/his visualisation pipeline. At the end of

this step a simple click in the ParaView interface will create a Python file that programmatically defines the visualisation operations to be performed *in-situ*.
4) Finally the real simulation is ran using a full resolution size mesh. The co-processing part of the simulation reads the python script containing the definition of the visualisation pipeline. This step is expected to be time-consuming.



(a) (b)

Fig. 3: (a) original geometry of our use case
(b) a final coprocessed picture of our simulation

6.2 Use Cases

Our simulations have been run on Ivanoe, an EDF corporate supercomputer, composed of 1382 nodes, each node having 12 cores for a total of 16584 cores. In these simulations we associate one MPI process by core and we use from 720 cores up to 3600 cores. We include two use cases that were run on this supercomputer. The choice of the cases is motivated by two main factors: the size of the mesh and the complexity of the visualization pipeline. We detail next these two impacting factors.
1) Mesh size. We chose to use two meshes representing the same geometry but at different resolutions, one made of 51M hexahedral elements and another of 204M hexahedrals. As we have already outlined in section 2, in our industrial environment, our simulation engineers are blocked when performing visualisation tasks on their PCs for meshes containing 50 M cells. We chose a 51M elements mesh as it is representative of a typical study at EDF which will induce performance issues for the visualization process. Furthermore, it more than doubles the size used in the results presented in [9] for the PHASTA adaptor. On the other hand, when research oriented simulations are performed at EDF, these simulations currently contain around 200M el-

ements. We choose this size as a typical research oriented or "heavy mesh" simulation data.
2) Pipeline complexity. We define two pipelines aimed to be representative of two different situations: users performing simple and light visualization operations (e.g. generating slices in a volume) and another using very time-consuming visualization tasks (e.g. performing volume rendering).

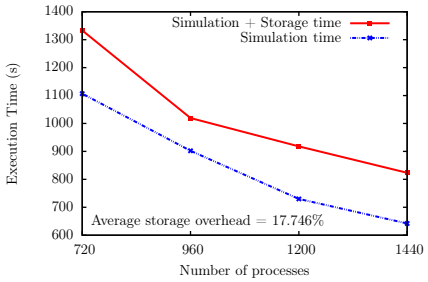
Table 2: Description of our two use cases.

USE CASES SUM UP			
NAME	SIZE	PIPELINE	FIGURES
CASE_A	51M hexahedrals, industrial size case	**heavy**: volume rendering, celldatatopointdata and glyphs	4a 4c 4e
CASE_B	204M hexahedrals, research size case	**light**: 9 slices, celldatatopointdata	4b 4d 4f

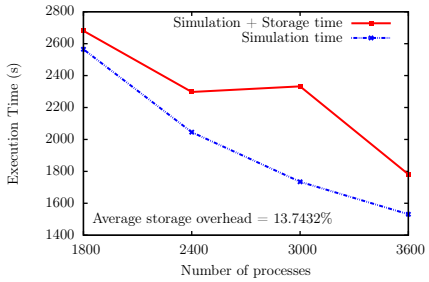
In the following we name our uses cases: *CASE_A*, use case using an average mesh size of 51M hexahedrals and a visualization pipeline including volume rendering which aims to be very time-consuming. *CASE_B*, our second use case, contains a light visualization pipeline simply performing some slices but on a large mesh of 204M hexahedrals.
Table 2 summarizes the composition of these use cases. In all our use cases we run a simulation consisting in a fluid with physical properties identical to water passing through the mesh. Then the output is generated at each step, for a total of 10 co-processed visualization images.

6.3 Results

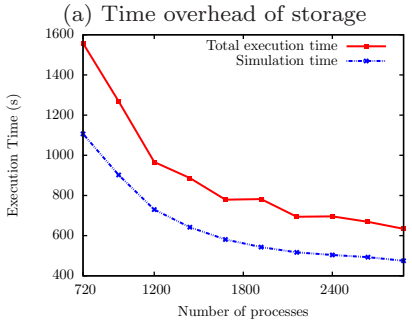
Figure 3b presents an image obtained from one of our *in-situ* simulations with *CASE_A*. We see the flux of water moving around the vertical cylinders, the glyphs being attached to the velocity vectorial field. The color of the volume rendering represents the turbulent viscosity of the fluid.
We establish first the overhead induced by storing our simulation results in figure 4a and 4b. We observe an average of 18% and 14% of time used to store results, for *CASE_A* and *CASE_B* respectively. These figures correspond to the comparison of T_w and $T_s + T_w$ in equation 1. This overhead tends to increase with the number of processes in use. One can also notice that the overhead is also not stable and subject to important variations with a peak at 26%. We thus identify the storage process as a bottleneck in everyday CFD studies for its average overhead and its high instability in execution time.
Figure 4c shows two graphs of *CASE_A*: in red the execution time versus the number of cores, in blue the execution time without the co-processing



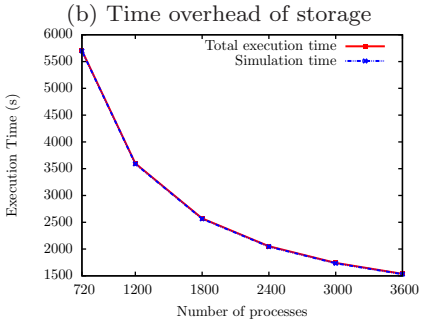
(a) Time overhead of storage



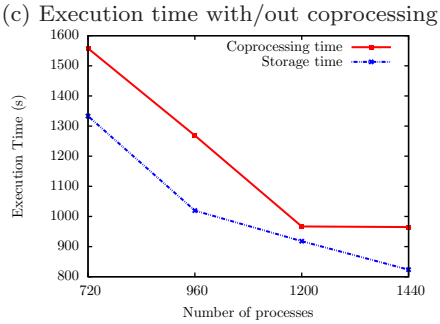
(b) Time overhead of storage



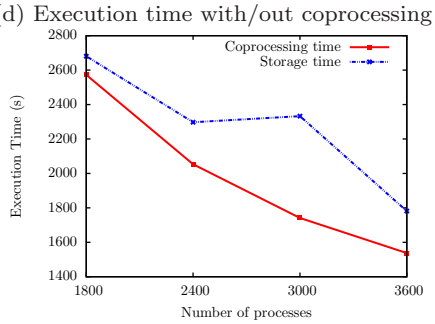
(c) Execution time with/out coprocessing



(d) Execution time with/out coprocessing



(e) Execution time comparison with storage.



(f) Execution time comparison with storage.

Fig. 4: CASE_A (left) and CASE_B (right) results

overload. These figures correspond to the comparison of T_s and $T_s + T_{process}$ in equation 2. We are satisfied by this overload that is comprised between 20 and 30% of the total execution time, when we chose complicated task with a high $T_{process}$. Moreover, it looks like this overload is decreasing when the number of cores increases. Figure 4d shows the exact same behavior in the $CASE\_B$ experiment. Both graphs are difficult to distinguish as the time needed for co-processing is circumscribed between 6 and 10 seconds, the overload (the difference between T_s and $T_s + T_{process}$) is less than one percent of the total execution time. We notice that having heavy $T_{process}$ is not very

usual in our applications and we consider $CASE\_A$ as an example of worst case scenario.
We also decided to compare the Catalyst overhead with a non-VTK-based storage strategy that performs no visualization operations. Figure 4e and 4f, show the comparison of the global execution time with Catalyst co-processing versus the Ensight Gold storage format. This means comparing $T_s + T_w$ and $T_s + T_{process} + T_{w-in-situ}$. Figure 4e presents our implementation results with $CASE\_A$. This compares positively for Catalyst as the overhead is approximately 10% and decreases when the number of cores increases. We notice that for $CASE\_A$ the heavy $T_{process}$ is already taken into account in the *in-situ* curve but $T_r + T_v$ is still not performed for the traditional visualisation scheme. This means that this result is very positive and we should not forget that $T_r + T_v$ is very time consuming for this case (and saturates our scientific PCs at EDF R&D).
Figure 4f presents our results for $CASE\_B$. Here we can see the potential of Catalyst when lighter and more relevant visualization tasks are processed. Indeed, there is no more overhead as we gain an average of 10% of execution time while freeing ourselves from storage issues (we evaluate the execution time peak of 3000 processes as a result of concurrent accesses on our supercomputer storage disks). To emphasize this, Table 3 shows how much data each solution generates, namely a basic storage in Ensight Gold format versus our co-processing implementation using Catalyst. These informations are those of our CASE_B when performing a 10 steps simulation. Both size are expected to grow proportionally to the size of the mesh input, and the number of steps. Therefore, we expect the gain provided by the use of co-processing to be increasingly interesting when moving forward in use case size.
We also studied in [11] the total memory used when running in-situ visualization compared to writing simulation results in Ensight Gold format. We observe that memory consumption is increased by an approximate factor varying from 2 to 3. This can be explained by both our first naive memory management approach and also by a natural increase in memory consumption when visualization operations are to be performed. Concerning the memory management Kitware recently released the so-called "zero-copy VTK" that can be used to avoid naive deep copy of the simulation data structures.

Table 3: CASE_B comparison between the size of processed results and simple storage. The simulation was run on 10 steps, with 10 pictures co-processed.

***PROCESSING SIZE COMPARISON**	
STORAGE	COPROCESSING
57Gio	1,3Mio

7 Conclusion

This chapter provides an overview of the efforts needed to transition a traditional simulation code to an *in-situ* model. We believe that the results of our experiments are bringing new insights to the community, especially to the simulation teams (including simulation code developers and the visualization experts they work with) that are considering the transition to *in-situ* processing.

The main finding presented in this chapter is that we have successfully integrated Catalyst into *Code_Saturne* (a computational fluid dynamics code developed at EDF R&D). Both Catalyst and *Code_Saturne* are Open Source software and this development can be downloaded, used or tested freely by everyone. After testing the prototype in our corporate supercomputer Ivanhoe, we found Catalyst to be a relevant solution to provide *Code_Saturne* users with visualization co-processing. Catalyst also allowed for a simple and fast implementation of a simulation code adaptor.

The results presented are based on a 51M and a 204M elements mesh, which is above the average size case used by EDF engineers in our industrial environment. We plan to perform simulations on at least 400M elements meshes in the near future, using the same supercomputer. We performed our simulations from 60 up to 300 nodes. This is due to the typical simulation node size in Ivanhoe being around 150 nodes for our engineers. We also plan to work on another of our corporate supercomputers, an IBM BG/Q with 65k cores. In that case, we will test on a much larger number of cores.

The increase of memory usage, described in the results section of [11], indicates that memory optimizations are to be performed before running on the IBM BG/Q. We did not, in this study, perform any delicate memory tweaking in order to reduce the memory consumption. We are currently working on this point, experimenting with the new VTK *in-situ* data structures implemented recently by Kitware, the so-called "zero copy VTK". This approach aims to facilitate the memory management in the adaptor without the use of complicated pointer manipulation; we expect to reduce memory overhead without much increasing code complexity.

Another ongoing development consists on how we deal with the ghost levels generated by *Code_Saturne*. Indeed, we want to use the same spatial partition of the meshes for *Code_Saturne* and Catalyst, the aim being not to degrade the execution time by "not necessary data exchanges" among MPI ranks. We currently use ParaView D3 filter (a filter originally performing a redistribution of the data among MPI processes) as a ghost cell handler. However, we asked Kitware for the integration in ParaView/Catalyst of a new filter to perform a direct generation of ghost cells from existing distributed data. This development has been finished in December 2013.

This chapter has been dedicated on how to deal with large data using visual co-processing but we are also testing the computational-steering capabilities of Catalyst, the so-called Live Catalyst. This currently allows the modifica-

tion of the ParaView pipeline parameters while the numerical simulation is running.
In conclusion, we are mostly satisfied with the integration of Catalyst in *Code_Saturne*. The first version of our integration will be released as part of a new version of this open-source software.

References

1. F. Archambeau, N. Mechitoua, and M. Sakiz. Code saturne: A finite volume code for the computation of turbulent incompressible flows. In *Industrial Applications, International Journal on Finite Volumes, Vol. 1*, 2004.
2. F. Archambeau, N. Mechitoua, and M. Sakiz. Edf, code_saturne version 3.0 practical user's guide, 2013.
3. J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali. Parallel computational steering for hpc applications using hdf5 files in distributed shared memory. *Visualization and Computer Graphics, IEEE Transactions on*, 18(6):852–864, 2012.
4. J. Brooke, P. Coveney, J. Harting, S. Pickles, Pinning, and A. R.L. Porter. Computational steering in realitygrid. *Proc. UK E-Science All Hands Meeting*, pages 885–888, 2003.
5. H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*, pages 191–198, 2005.
6. K. D. M. David Thompson, Nathan D. Fabian and L. G. Ice. Design issues for performing in situ analysis of simulation data. In *Technical Report SAND2009-2014, Sandia National Laboratories*, pages 7–18, 2009.
7. D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom. Concurrent visualization in a production supercomputing environment. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):997–1004, 2006.
8. A. Esnard, N. Richart, and O. Coulaud. A steering environment for online parallel visualization of legacy parallel simulations. In *Distributed Simulation and Real-Time Applications, 2006. DS-RT'06. Tenth IEEE International Symposium on*, pages 7–14, 2006.
9. N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96, 2011.
10. C. Law, A. Henderson, and J. Ahrens. An application architecture for large data visualization: a case study. In *Parallel and Large-Data Visualization and Graphics, 2001. Proceedings. IEEE 2001 Symposium on*, pages 125–159, 2001.
11. B. Lorendeau, Y. Fournier, and A. Ribes. In-situ visualization in fluid mechanics using catalyst: A case study for code saturne. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 53–57, Oct 2013.
12. A. Ribes and C. Caremoli. Salome platform component model for numerical simulation. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 553–564, 2007.
13. R. B. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland. Visualization and parallel i/o at extreme scale. *Journal of Physics: Conference Series*, 125(1):012099, 2008.
14. T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. O'Hallaron. From mesh generation to scientific visualization: An end-to-end

approach to parallel supercomputing. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 12–12, 2006.

15. P. Vezolle, J. Heyman, B. D'Amora, G. Braudaway, K. Magerlein, J. Magerlein, and Y. Fournier. Accelerating computational fluid dynamics on the ibm blue gene/p supercomputer. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 159–166, 2010.
16. B. Whitlock, J. M. Favre, and J. S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, EG PGV'11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.