Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Implementing a XDMF/HDF5 Parallel File System in Alya

Raúl de la Cruz[a,*], Hadrien Calmet[a], Guillaume Houzeaux[a]

[a]Barcelona Supercomputing Center, Edificio NEXUS I, c/ Gran Capitán 2-4, 08034 Barcelona, Spain

Abstract

Alya is a Computational Mechanics (CM) code developed at Barcelona Supercomputing Center, which solves Partial Differential Equations (PDEs) in non-structured meshes, using Finite Element (FE) methods. Being a large scale scientific code, Alya demands substantial I/O processing, which may consume considerable time and can therefore potentially reduce speed-up at petascale. Consequently, I/O task turns out a critical key-point to consider in achieving desirable performance levels. The current Alya I/O model is based on a master-slave approach, which limits scaling and I/O parallelization. However, efficient parallel I/O can be achieved using freely available middleware libraries that provide parallel access to disks. The HDF5 parallel I/O implementation shows a relatively low complexity of use and a wide number of features compared to others implementations, such as MPI-IO and netCDF. Furthermore, HDF5 exposes some interesting aspects such as a shorter development cycle, a hierarchical data format with metadata support and is becoming a *de facto* standard as well. Moreover, in order to achieve an open-standard format in Alya, the XDMF approach (eXtensible Data Model Format) has been used as metadata container (light data) in cooperation with HDF5 (heavy data). To overcome the I/O barrier at petascale, XDMF & HDF5 have been introduced in Alya and compared to the original master-slave strategy. Both versions are deployed, tested and measured on Curie and Jugene Tier-0 supercomputers. Our preliminary results on the testbed platforms show a clear improvement of the new parallel I/O compared with the original implementation.

1. Introduction

Alya is a general Computational Mechanics (CM) research code developed at Barcelona Supercomputing Center that solves Partial Differential Equations (PDEs) in non-structured meshes. It covers a wide range of physical problems, such as: compressible and incompressible flows, large strain solid mechanics, thermal flows, excitable media or quantum mechanics for transient molecular dynamics. Alya is written from scratch as a parallel code using a hybrid strategy: MPI/OpenMP. On one hand, mesh partitioning is automatically done using METIS, creating a set of subdomains that communicates via MPI. On the other hand, internal loops are parallelized using OpenMP to take profit of memory sharing in multicore clusters. This paper describes the main ideas implemented behind the parallel I/O aspects. The different strategies are illustrated through the solution of numerical examples.

Large scale scientific applications demand huge dataset storage, which may consume a large amount of the simulation execution time and can reduce drastically the speed-up. This fact shows that I/O could become a critical aspect to achieve desirable performance levels for some particular scientific problems. The first and direct approach to the problem for a parallel code such as Alya is based on sending data to the MPI master task, which is responsible of writing data directly into disk through I/O POSIX calls. As though, this model is not scalable due to its master-centric philosophy, and it does not permit I/O parallelization.

On the other hand, an efficient parallel I/O can be implemented with some available libraries that supply parallel access to disks. Ordered from major-to-minor complexity and performance, and from minor-to-major features the most known implementations are: MPI-IO, netCDF and HDF5. Despite exhibiting a slightly worse performance [1, 2, 3], the HDF5 library has been finally chosen in this work due to several reasons. First, it can be deployed in the CM code in a short time, reducing the development cycle. Second, unlike netCDF, HDF5 supports hierarchical structures in data files, useful to organize in a logical way the generated mesh data. Third, its parallel I/O implementation is build over MPI-IO, allowing for substantial performance gains due to collective I/O optimizations. As a final remark, it is almost considered a standard in parallel I/O.

This work is organized as follows: Section 2. introduces the concept of XDMF as a lightweight container for scientific data and its advantages. Section 3. briefly reviews the HDF5 features and its internals. Next, Section 4.

*Corresponding author.
tel. +0-000-000-0000 fax. +0-000-000-0000 e-mail. delacruz@bsc.es

proposes several strategies considered to implement a parallel I/O service for Alya which includes XDMF & HDF5 formats. Finally, in Section 5. we present a comparative result for Alya in Jugene [4] and Curie [5], large supercomputers based on BlueGene/P and Intel Nehalem-EX nodes respectively. The comparison is carried out between the original I/O mechanism and the new XDMF/HDF5 implementation.

2. Introducing XDMF

The eXtensible Data Model and Format (XDMF) were created to standardize the scientific data between High Performance Computing (HPC) codes. Originally, it was developed by US Army Research Laboratory (ARL) for Lawrence Livermore National Laboratory (LLNL), CTH and the SIERRA framework code collection from Sandia National Laboratory (SNL) [6, 7]. Furthermore, most modern visualization software tools, like *Paraview* [8], are capable of reading XDMF postprocessed data.

XDMF views data as composed of two basic types: **Light** data and **Heavy** data. The former can contain both metadata and small amounts of values. The latter typically consists of large datasets. In order to store the Heavy data, the HDF5 format is used. XDMF uses eXtensible Markup Language (XML) to store Light data and to describe the shape of the large data sets. The XML format is widely used for many purposes. It is case sensitive and is made of three major components: elements, entities and processing information. Each element begins with a <tag> and ends with a </tag>. Futhermore, the element can have a name attribute and contain one or more domain elements (computational domain).

A **Domain** can have one or more **Grid** elements. Each Grid (mesh) contains a **Topology**, a **Geometry** and zero or more **Attribute** elements. The Topology represents the connectivity information of the mesh whereas the Geometry conveys the coordinates of the element nodes. Finally, the Attribute element specifies values such as scalars and vectors, which can be centered at node, edge, face, cell or grid of the mesh.

To bring information for connectivity, geometry or attributes, XDMF defines a **DataItem** element. A DataItem can provide the actual values directly or provide an external physical storage (e.g. HDF5). There are six different families of DataItems. Ordered from lower to higher data structure complexity they are: **uniform**, **collection**, **Tree**, **Hyperslab**, **Coordinates**, **function**. The following markup describes a minimal example of XDMF with a single grid containing its topology, geometry and attribute entities.

```
<Xdmf>
 <Domain>
  <Grid>
   <Topology>
   </Topology>
   <Geometry>
   </Geometry>
   <Attribute>
   </Attribute>
  </Grid>
 </Domain>
</Xdmf>
```

3. HDF5 in a nutshell

NCSA has developed the last version 5 of its Hierarchical Data Format (HDF5) library [9], a re-design of the previous version with some extra features. Some of the new features are: support for large files (> 2 GBytes) and large number of objects, compression packages support or the implementation of a parallel I/O implementation through MPI-IO among many others.

Basically, HDF5 provides a portable (endianess) and flexible file format and a API to store or retrieve multidimensional arrays, which suites pretty well for scientific data, available on most important HPC platforms. Many special purpose scientific projects use HDF5 as a solution for dataset storage. Some numerical codes of renown using it are: Cactus, Chombo, RAMS or Ensight.

The support for parallel data access in HDF5 is built on top of MPI-IO library, which ensures its portability since MPI-IO has become a *de facto* standard for parallel I/O. Additionally, the parallel HDF5 uses a tree-like file structure similar to the UNIX file system, where data is sparsely arranged using different types of blocks (super, header, data and extended). To manage this irregular layout pattern, the parallel HDF5 uses dataspaces and hyperslabs, which define the data organization, map and transfer data between memory space and the file space.

Every HDF5 file is organized following a hierarchical structure, containing two primary structures with support of metadata information: groups and datasets. The former can contain zero or more instances of HDF5 groups or datasets structures. On the other hand, the latter structure contains a multidimensional array of data elements. Each group or dataset may have an associated attribute list, which provides extra information about the object. The metadata attributes can describe a specific nature or use of the group or dataset. Regarding the hierarchical structure, managing groups and datasets is very similar to a UNIX filesystem with directories and files, where a root element (a group in HDF5) and several directories (others groups) or files (datasets in HDF5) exist. Figure 1 shows an example of a hierarchical HDF5 file with groups and datasets.

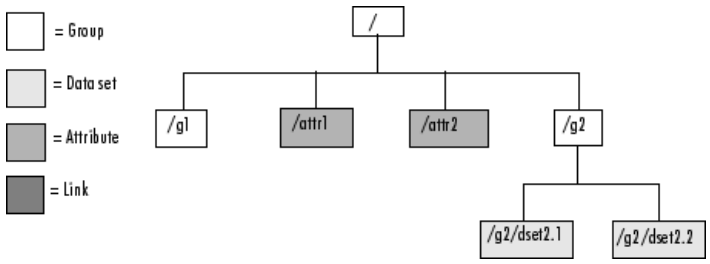

Fig. 1. HDF5 hierarchical example. This H5 file contains groups, attributes, datasets and links to other objects. All the information dangle from the root object (/) of the H5 file.

4. Parallel I/O implementation in Alya

The parallel I/O implementation was carried out supporting ours needs. A Computational Fluid Dynamic (CFD) simulation generates a lot of datasets, which may include many variables and/or many time steps. The postprocess step could be different in sense of the user needs. For instance, the user may want to see the distribution of the problem domain among the different processors or just the evolution of a flow in a cavity test problem. For this reason, two arrangement distributions of data are supported by the parallel implementation. It is also important to remark that the master is responsible of writing XDMF files (light data), whereas the slaves are in charge of the heavy data creation in parallel (using parallel HDF5).

In the following subsections, the different strategies taken for XDMF (light data) and HDF5 (heavy data) formats are described and their advantages and disadvantages explained in detail.

4.1. XDMF Strategy

Two interesting strategies to arrange data in XDMF format for CFD codes are at least devised. Ordered from lower to higher complexity of implementation, we have: one dataset space per unknown variable and multiple dataset spaces per each problem domain. In addition, depending on the simplicity of implementation, the former strategy supports two ways of writing data, repeating and not repeating unknown variables values at boundary nodes. The advantages and disadvantages of each strategy are discussed next.

The first scheme to describe the light data was designed to obtain a temporal evolution of the attribute values along the simulation in only one dataset space. To achieve this scheme, one grid of type **Collection** with **Temporal** as the collection type must be added to the XDMF file. This collection is an array of **Uniform** grids where each grid describes the attributes at every time step. Some examples of attributes are the pressure or the velocity at nodal points. To set the time iteration at each grid, the **Time** tag is used. For each element of the uniform grid array, we specify the time as **Single** type and their value as the parameter.

Inside the **Uniform** grid, the organization of the light data is straightforward. Each inner grid contains a **Topology** and a **Geometry** entity to describe the mesh at that iteration. For the sake of simplicity, the topology type is **Mixed**, which means that no considerations must be taken concerning the type of the elements in the mesh. Tetrahedrons, hexahedron, pyramids, wedges and others element types can be mixed in the connectivity dataset by specifying in the heavy data file the element type before each nodes id list. Besides, the **Geometry** entity is in **XYZ** format, meaning that the coordinates are stored in this axis order for each node of the domain. Finally, an **Attribute** entry is added in the **Uniform** grid in order to convey every variable of the computational domain.

The following markup snippet shows an example of this strategy for a cavity test problem, where the velocity is gathered at the nodal points of the mesh in two different timesteps. Note that some parameters (*Dimensions* and *NumberOfElements*) have been set to their theoretical values instead of their literal values which must be computed during the simulation run.

```
<Xdmf>
  <Domain>
    <Grid GridType="Collection" CollectionType="Temporal">

      <Grid Name="cavtet4" Type="Uniform">
        <Time Type="Single" Value="0.10000" />
        <Topology Type="Mixed" NumberOfElements="nelem-total">
          <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-total" NumberType="Int" Format="HDF" >
            cavtet4-mesh.h5:/CONNE
          </DataStructure>
        </Topology>
        <Geometry GeometryType="XYZ">
          <DataStructure Dimensions="3 npoin-total" NumberType="Float" Presicion="8" Format="HDF" >
            cavtet4-mesh.h5:/COORD
          </DataStructure>
        </Geometry>
        <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
          <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="3 npoin-total">
```

```
          cavtet4-0000000.h5:/VELOC
        </DataStructure>
      </Attribute>
    </Grid>

    <Grid Name="cavtet4" Type="Uniform">
      <Time Type="Single" Value="0.20000" />
      <Topology Type="Mixed" NumberOfElements="nelem-total">
        <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-total" NumberType="Int" Format="HDF" >
          cavtet4-mesh.h5:/CONNE
        </DataStructure>
      </Topology>
      <Geometry GeometryType="XYZ">
        <DataStructure Dimensions="3 npoin-total" NumberType="Float" Presicion="8" Format="HDF" >
          cavtet4-mesh.h5:/COORD
        </DataStructure>
      </Geometry>
      <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
        <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="npoin-total">
          cavtet4-0000001.h5:/VELOC
        </DataStructure>
      </Attribute>
    </Grid>

  </Grid>
 </Domain>
</Xdmf>
```

The second strategy arranges data in multiple dataset spaces. The main advantage of this method is that it allows the user to visualize results on computational domains independently along the time simulation. This arrangement is more suitable for HPC simulations, where thousands of processors are used and the user may have some interest into domain decomposition profiling. However, this strategy makes the light and heavy data files longer because it includes redundant data (boundary nodes of each domain are repeated) along the grid. In order to generate the XDMF markup file, a temporal evolution of the attributes involved in the simulation must be designed. Therefore, one grid tag of type **Collection** is added, where **Temporal** is used as the collection type. As in the previous strategy, this collection is an array of **Uniform** grids where each grid describes the attributes at every time step.

Finally, an outer collection of **Spatial** grids is required to describe the computational domains involved in the simulation. At this time, the external collection is an array of different grid domains of temporal type, which includes several instances of the previously defined temporal grid collection.

The following XDMF markup shows an example of this strategy. Note that it contains as many temporal grid collections as computational domain we have in the simulation, and as many uniform grids as time iterations are computed and written.

```
<Xdmf>
 <Domain>
  <Grid GridType="Collection" CollectionType="Spacial">
   <Grid GridType="Collection" CollectionType="Temporal">
    <Grid Name="cavtet4" Type="Uniform">
     <Time Type="Single" Value="0.10000" />
     <Topology Type="Mixed" NumberOfElements="nelem-dom1">
      <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-dom1" NumberType="Int" Format="HDF" >
       cavtet4-mesh.h5:/CONNE/DOM-1
      </DataStructure>
     </Topology>
     <Geometry GeometryType="XYZ">
      <DataStructure Dimensions="3 npoin-dom1" NumberType="Float" Presicion="8" Format="HDF" >
       cavtet4-mesh.h5:/COORD/DOM-1
      </DataStructure>
     </Geometry>
     <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
      <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="3 npoin-dom1">
       cavtet4-0000000.h5:/VELOC
      </DataStructure>
     </Attribute>
    </Grid>
    <Grid Name="cavtet4" Type="Uniform">
     <Time Type="Single" Value="0.20000" />
     <Topology Type="Mixed" NumberOfElements="nelem-dom-1">
      <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-dom1" NumberType="Int" Format="HDF" >
       cavtet4-mesh.h5:/CONNE/DOM-1
      </DataStructure>
     </Topology>
     <Geometry GeometryType="XYZ">
```

```
          <DataStructure Dimensions="3 npoin-dom1" NumberType="Float" Presicion="8" Format="HDF" >
            cavtet4-mesh.h5:/COORD/DOM-1
          </DataStructure>
        </Geometry>
        <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
          <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="3 npoin-dom1">
            cavtet4-0000001.h5:/VELOC/DOM-1
          </DataStructure>
        </Attribute>
      </Grid>
    </Grid>

    <Grid GridType="Collection" CollectionType="Temporal">
      <Grid Name="cavtet4" Type="Uniform">
        <Time Type="Single" Value="0.10000" />
        <Topology Type="Mixed" NumberOfElements="nelem-dom2">
          <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-dom2" NumberType="Int" Format="HDF" >
            cavtet4-mesh.h5:/CONNE/DOM-2
          </DataStructure>
        </Topology>
        <Geometry GeometryType="XYZ">
          <DataStructure Dimensions="3 npoin-dom2" NumberType="Float" Presicion="8" Format="HDF" >
            cavtet4-mesh.h5:/COORD/DOM-2
          </DataStructure>
        </Geometry>
        <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
          <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="3 npoin-dom2">
            cavtet4-0000000.h5:/VELOC/DOM-2
          </DataStructure>
        </Attribute>
      </Grid>
      <Grid Name="cavtet4" Type="Uniform">
        <Time Type="Single" Value="0.20000" />
        <Topology Type="Mixed" NumberOfElements="nelem-dom2">
          <DataStructure Dimensions="(1+nnode(ltype(ielem)))*nelem-dom2" NumberType="Int" Format="HDF" >
            cavtet4-mesh.h5:/CONNE/DOM-2
          </DataStructure>
        </Topology>
        <Geometry GeometryType="XYZ">
          <DataStructure Dimensions="3 npoin-dom2" NumberType="Float" Presicion="8" Format="HDF" >
            cavtet4-mesh.h5:/COORD/DOM-2
          </DataStructure>
        </Geometry>
        <Attribute Name="VELOC" Center="Node" AttributeType="Vector">
          <DataStructure Format="HDF" DataType="Float" Precision="8" Dimensions="3 npoin-dom2">
            cavtet4-0000001.h5:/VELOC/DOM-2
          </DataStructure>
        </Attribute>
      </Grid>
    </Grid>
  </Grid>
 </Domain>
</Xdmf>
```

4.2. HDF5 Strategy

HDF5 postprocessing support has been implemented in Alya as a service, allowing the user to use either the parallel I/O feature or the sequential one, where the master MPI task writes all data directly to disk.

Given that the master in Alya is a non-computational task (it does not compute any dependent mesh result), only slaves MPI tasks belong to the HDF5 communication group. Thus, if the execution problem has N tasks, only $N-1$ tasks read and write the datasets. The implementation strategy for the HDF5 service takes into consideration the two strategy methods described in Section 4.1. (see Figures 2 and 3). Two groups of datasets have been considered when mapping to HDF5 format:

- The first group is the mesh datasets, which defines topology and geometry (nodes connectivity and coordinates) for futher postprocessing. Parallel I/O on mesh data must be considered to enable parallel postprocessing of the results through visualization tools. As commented before, two HDF5 datasets are generated in this group: the coordinates (linked with *Geometry* entity in XDMF), which contains the `npoin` coordinates (x,y,z) of the nodes, and the connectivity (linked with *Topology* entity in XDMF), which relates the `nelem` elements with their respective nodes. In the case of the coordinates, boundary nodes are repeated among shared domains, so the coordinates dataset in HDF5 format contains more nodes than the original one. Obviously, the connectivity should be consistent with this *new* global mesh, having more nodes than the original one. One important advantage of this format is that, parts of the

mesh corresponding to the METIS partition can be extracted easily in postprocessing, but also avoiding additional communication to coordinate which domain writes shared node information.

- The second group is the solution dataset, which includes the nodal variables that are computed in Alya. As in the case of the coordinates, these files have more nodal values than the original mesh. In addition to nodal values, elemental values can also be needed, typically computed at the Gauss points of the elements (internal variables in solid mechanics and subgrid scale in fluid mechanics).

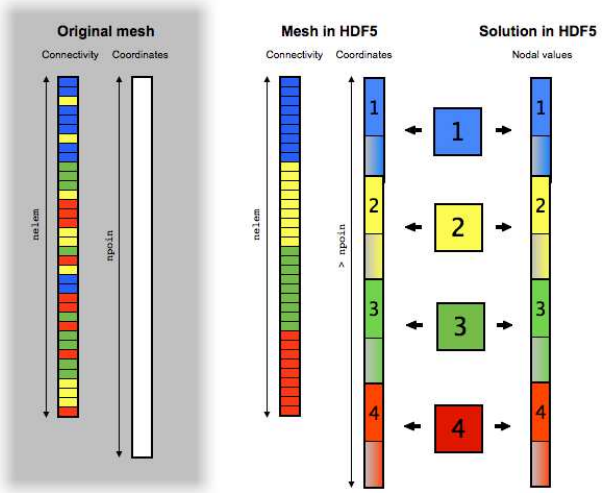

Fig. 2. HDF5 mapping in Alya. From an original mesh, new H5 mesh datasets are generated where boundary nodes information is duplicated.

5. Performance results

A complex human respiratory airways mesh has been used to benchmark the original I/O system versus the new XDMF/HDF5 service (see Figure 4.Left). This hybrid element mesh with prisms and tetrahedrons solves the incompressible Navier-Stokes equations using a fractional step like technique [10]. This test has been deployed and run on both testbed platforms (Jugene and Curie), coming up with three different granularities: 4M, 17M, and 27M element mesh. Tables 1.Left and 1.Right describe the testbed platforms and the problem sizes respectively.

System name	Curie	Jugene
Architecture	Intel Xeon Core i7 X7560 (Nehalem-EX)	IBM BlueGene/P PowerPC 450
Chips × Cores	8 × 4	1 × 4
Clock	2.26 GHz	850 MHz
SP GFlops†	93.76 (SSE2)	13.6
DP GFlops†	46.88	13.6
L1 Cache (D+I)	32 kB + 32 kB	32 kB + 32 kB
L2 Cache	256 kB per core	1920 B per core
L3 Cache	24 MB (shared)	2 x 4 MB (shared)
Main memory	24 GB	2 GB
Bandwidth	32 GB/s	13.6 GB/s
Watts x hour	130	39
Compiler	Intel/GCC Compiler (v12.0.1/v4.4.4)	IBM XL Compiler (v9.0/v11.01)

Test name	airways-4M	airways-17M	airways-24M
Type of test	I/O	I/O	I/O
Subdomains	255	511	1024
Total nodes (inter+bound)	1210430	17426196	27873551
Min. nodes	644 (78)	6698 (158)	5462 (597)
Max. nodes	1218 (155)	9411 (254)	10660 (694)
Ave. nodes	998	7259	7352
Min. elements	1874 (62)	23878 (222)	11760 (597)
Max. elements	2771 (77)	67761 (116)	55760 (695)
Ave. elements	2113	35793	26647

Table 1. Left: Architectural configuration of the platforms used in our tests. †Only one core is considered. Right: Partition summary for Jugene and Curie tests. Domains with minimum, maximum and average values are shown in parenthesis.

In order to gather an accurate information of the timing and the MPI communication profiling, Extrae [11] and Paraver [12] tools have been used to instrument and extract such data. Figure 4.Right shows a Paraver trace of an execution of the 27 million element nasal mesh. Table 2 shows a summary of the postprocessing times obtained with Paraver on master and slaves tasks for the human respiratory model described above. This test writes the velocity vector $\vec{v} = \{v_x, v_y, v_z\}$ and the pressure at each node of the mesh and also the geometry of the model. The amount of data stored per snapshot iteration is around 141MBytes and 196MBytes in the small case (4M element mesh) and 541MBytes and 770MBytes in the big case (27M element mesh) for nodal variables and mesh data respectively.

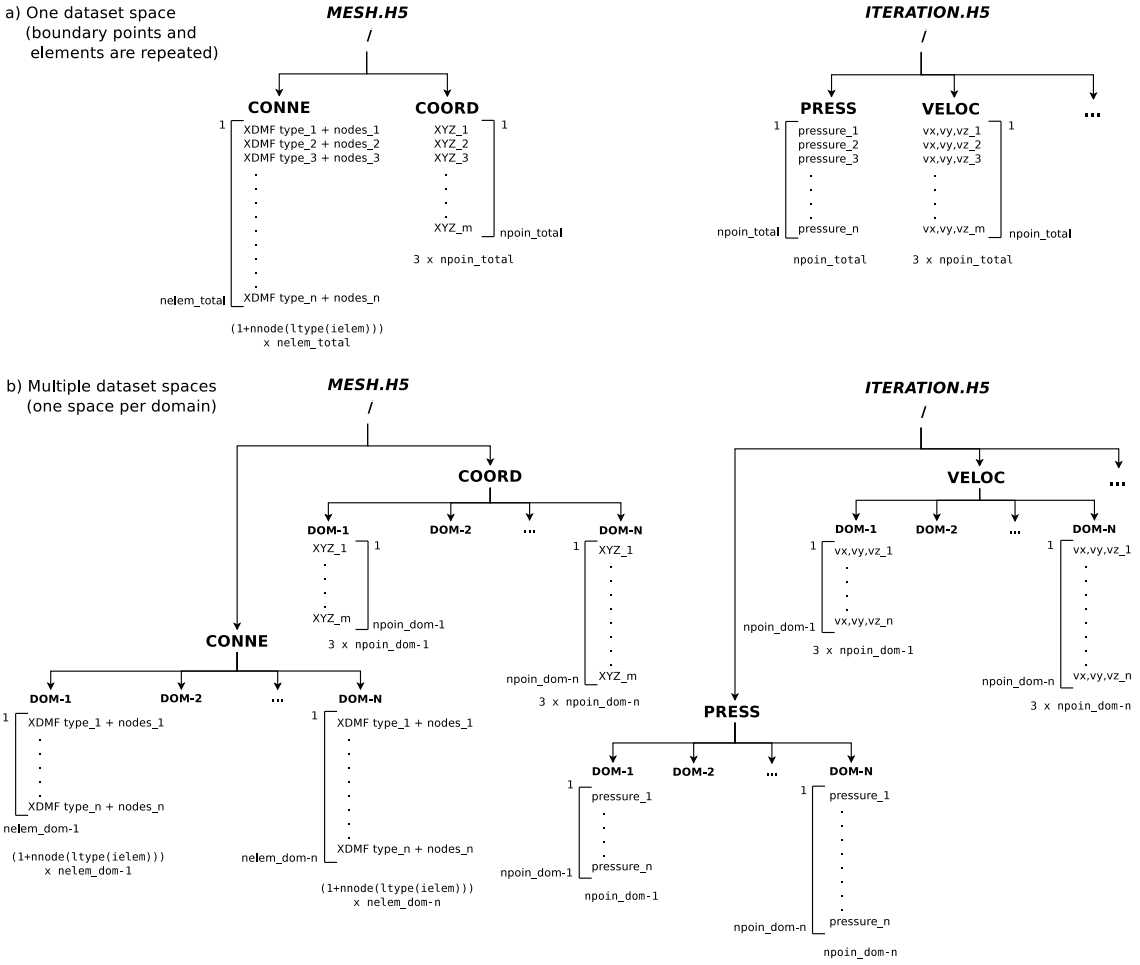

Fig. 3. a) HDF5 representation of one dataset space strategy. In this case, boundary nodes are repeated among neighbor domains. b) HDF5 representation of multiple dataset spaces strategy. As many dataset spaces dangle from each attribute (connectivity, coordinates, pressure, velocity, etc.) as computational domains were created during the Alya execution.

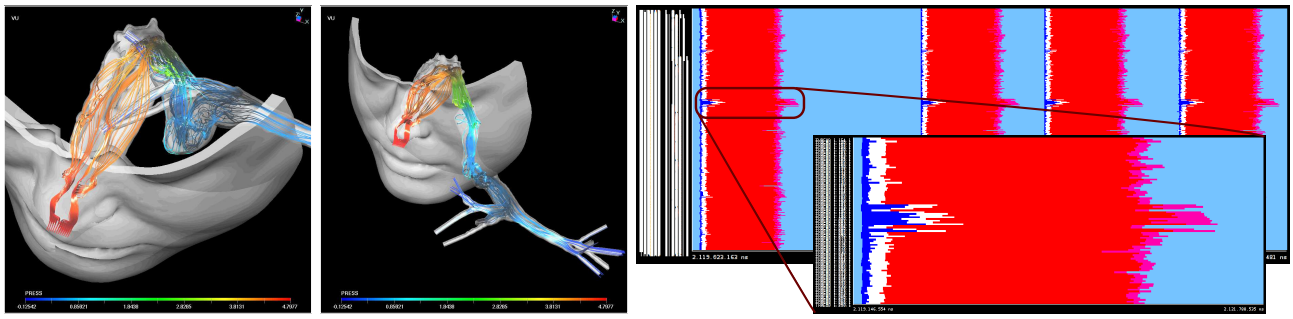

Fig. 4. Left: Respiratory airflow mesh. Right: Timeline of a Paraver trace section for the respiratory system run. Each row represents a MPI task, and every color a different state of the task.

Reviewing the results, we observe that in the original implementation, the master task is consuming the time on three states: running (copying data buffers of received data for further writing), `MPI_Recv` and I/O (writing directly the postprocessing data to disk). On the other hand, slaves spend most of the time in `MPI_Send` (blocked to obtain their turn to send data to master) and `MPI_Barrier` (waiting master task to finish I/O to continue to the next iteration of the solver loop). However in the HDF5 runs, the master task is idle and waits for the slaves to perform their MPI-IO calls through the HDF5 library. Comparing times of both implementations, it is shown a clear advantage of the HDF5 implementation, being 1.57x and 3.43x faster than the original version.

6. Conclusion

In this paper, the parallel I/O implementation of a general CM code, Alya, is described and briefly assessed in two Tier-0 supercomputers: Curie (Intel Nehalem-EX cluster) and Jugene (BG/P cluster). Then, the postprocess

		Curie		Jugene	
	Tasks	Master	Slaves	Master	Slaves
Master writes I/O	Running	103.6	2.2	112.1	1469.8
	`MPI_Recv`	69.7	–	5149.2	–
	`MPI_Send`	–	551.8	–	18160.5
	I/O (Posix I/O)	547.9	–	15735.1	–
	`MPI_Barrier`	0.07	164.4	1.21	187.8
	Total time (ms)	719.3	718.4	22824.1	19818.3
XDMF HDF5	I/O (MPI-IO)	–	438.1	–	6591.4
	`MPI_Barrier`	454.8	18.6	6641.8	48.6
	Total time (ms)	454.8	456.7	6641.8	6640.0
	Speed-up	1.57		3.43	

Table 2. Comparison of postprocessing times for the human respiratory model in one snapshot using master-centric versus XDMF/HDF5 model. Note that only the best case is shown for each architecture.

strategy implemented in Alya is tested on those platforms using a human respiratory system mesh of different element sizes. The gain in postprocess is quite significant when compared with the sequential old strategy, achieving results of 1.57 and 3.43 times faster. We have also shown that XDMF and HDF5 formats are pretty well suited to speedup I/O tasking in CFD codes.

Future research will be focused into explore new strategies to store light and heavy data in order to speedup the I/O task and include new visualization features for end-users. For instance, one interesting research line will be to design an intelligent I/O strategy to allow coupling the simulation execution with the visualization process *in situ*. This feature will permit to visualize simulation in real-time and abort the run if results are not behaving as the user expect, leading to significant computational and storage savings.

Acknowledgments

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557.

References

1. A. Jackson, F. Reid, A. Soba, and X. Sáez, "High performance i/o," *PDP 2011 - The 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2011.
2. J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *In Proceedings of Supercomputing*, 2003.
3. C. M. Chilan, M. Yang, A. Cheng, and L. Arber, "Parallel i/o performance study with hdf5, a scientific data package," *TeraGrid 2006: Advancing Scientific Discovery*, 2006.
4. "Jugene supercomputer." [Online]. Available: http://www.fz-juelich.de/jsc/jugene
5. "Curie supercomputer." [Online]. Available: http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm
6. "Xdmf standard." [Online]. Available: http://www.xdmf.org
7. l. J. Clarke and E. Mark, "Enhancements to the extensible data model and format (xdmf)," in *Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, ser. HPCMP-UGC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 322–327. [Online]. Available: http://dx.doi.org/10.1109/HPCMP-UGC.2007.30
8. "Paraview visualization tool." [Online]. Available: http://www.paraview.org
9. M. Yang, "Parallel hdf5 tutorial," in *14th IBM System Scientific User Group(ScicomP)*, 2008.
10. G. Houzeaux, R. Aubry, and M. Vázquez, "Extension of fractional step techniques for incompressible flows: The preconditioned Orthomin(1) for the pressure Schur complement," *Submitted to Comput. & Fluids*, 2009.
11. "Extrae instrumentation package." [Online]. Available: http://www.bsc.es/performance_tools
12. "Paraver, the flexible analysis tool." [Online]. Available: http://www.bsc.es/paraver