Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A Scalable Mesh Format for Parallel Unstructured Meshes in Scientific Applications

Ulrich Huber

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

A Scalable Mesh Format for Parallel Unstructured Meshes in Scientific Applications

Ein skalierbares Gitterformat für parallele, unstrukturierte Gitter in wissenschaftlichen Anwendungen

Author	Ulrich Huber
Supervisor	Prof. Dr. Michael Georg Bader
Advisor	Sebastian Rettenberger, M. Sc.
Date	March 15, 2017

Informatics V
Chair for Scientific Computing

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, March 15, 2017

Signature

Abstract

In this thesis, PXDMF is developed, a new scalable format to store parallel unstructured meshes. PXDMF includes a library, which supports reading and writing the new format. A initialized mesh is stored within a traversable data structure, which using applications can access. This new library and format is compared to the old format, it will replace, used between preprocessing applications and SeisSol, a simulation software for earthquakes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many large scale simulation applications the main focus of optimization is on fast and scalable solvers. Some of these applications use an unstructured mesh for discretization. In such applications, the development of the data format for the mesh, on which the simulation runs, is often neglected in comparison to the rest of the code. This results in a less optimized format and therefore greater file sizes and slower initialitzation. For simulations the I/O operations on the unoptimized format can prevent scalability. Therefore the data format should be as highly optimized as the rest of the code. To shorten the amount of time required to initialize the internal data structure, the library for the format should provide a data structure that can either be directly used by the application or can be easily transformed to the one desired. Furthermore the file size should be kept as low as possible, to ensure fast I/O operations and simplify the workflow. Additionally a mesh saved in such a format should be viewable by commonly used visualization tools. Finally for simulations, distributed over multiple communicating nodes, the location of data inside the format is important. As described by Rettenberger et al. in their paper, parallel access can be fastened by providing partitions of elements as single blocks of data [1]. Which means for partitioned unstructured meshes, that each node has to read only the elements of its partition and can ignore the rest of the elements, in contrast to attaching the partitioning information to the unsorted elements and each node iterating over all of them.

Assessing the above expectations of a partitioned format for unstructured meshes, multiple conflicts occure. Storing partitions as single blocks can have a negative impact on the file size. Since each partition is only reading its own block of data, elements shared between multiple partitions, need to be stored in the data block of each one. Hence the file size is increased by those elements, stored multiple times within the file. For large mesh sizes the generation of the file needs to be distributed over multiple nodes. Each node provides a set of cells and vertices required for the mesh. Depending on the mesh generator or mesh source, the given vertices and cells can be arbitraryly distributed.

Therefore a communication between the nodes needs to take place, to redistribute the vertices and cells to form the partitions, which enlarges the time required for writing the mesh. Furthermore for a small file size it is also necessary, to ensure that only required elements are saved for each partition, what further enlarges the generation time. The three expectations *fast initialization*, *file size* and *fast generation* create a magic triangle of formats for parallel unstructured meshes by mostly excluding each other.

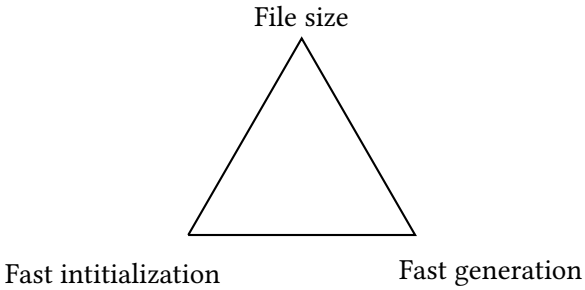


Figure 1.1: Conflict between expectations of a partitioned mesh format

While this magic triangle exists, the expectations are differnetly weighted, influencing the desired result. For example the partitioning of the data, for the reading of the file, is quite important, while the time required for the generation is insignificant in comparison. This stems from the fact, that meshes are often only generated once but used multiple times for simulations. Therefore optimizing the initialization, by providing partitions as single chunks of data, can be prioritized at cost of the generation time. The same applies to the file size, since partitioned files include only small portitions of duplicated data in comparison to the overall data. Therefore the file size is still quite small, even with sorting of the data into single blocks per partition.

1.1 Goals of the Thesis

SeisSol [2], developed by TUM and LMU, is a software package for simulating earthquakes by using wave propagation and dynamic rupture simulations. It solves its equations on a unstructured mesh of tetraheda, which allows approximations of complex 3D models and rapid initialization [2]. To replace an old format, with large file sizes and no ability to be visualized by software such as ParaView [3], a new format, as outlined above, is required.

The format will be used at the interface the between preprocessing software PUMGen [4] and SeisSol. Therefore the goal of this thesis is primarily developing a format and a library for generation and reading. To make the library useful for other applications, it should not be limited to usecases of SeisSol. Therefore expectations to the format and

the corresponding library are as follows:

- Generalization for different types of geometry
- Scalability concerning:
  - File size
  - Generation time
  - Initialization time
  - Number of used nodes
- Viewability by visualization tools

Additionally a test application is developed which reads the old and new file format and initializes a simple data structure on which a small simulation can be run. This is done, to benchmark both the old and new format for a comparison.

1.2 Outline

This thesis gives an in depth explanation of the developed format "Partitioned eXtensible Data Model and Format" (PXDMF) and its library. Chapter 2 starts with an evaluation of the old format and a following description of the formats used as basis of PXDMF. At the end of this chapter a breakdown of the internal structure of the format is given. Chapter 3 describes the structure of the data provided by the library and the algorithms used for reading and writing the format. PXDMF is evaluated in Chapter 4 by comparing it with the previously used format. The main points of comparison are the time needed for generation and initialization, as well as the file size. In Chapter 5 the results of the previous chapters are gathered and a conclusion is reached.

Chapter 2

Format

PXDMF is based on the eXtensible Data Model and Format 3 (XDMF) [5] and HDF5 [6]. XDMF is a schema for XML, which is supported by various viewing softwares such as ParaView [3] and VisIt [7]. Since storing great portions of data in XML files is inefficient, XDMF stores the mesh in HDF5 or binary files, while providing information on how this data is to be used in the XML file. In the documentation of XDMF this is generally described as seperating the data into light (the structure) and heavy (the data) parts [5].

The meshes which are saved in PXDMF, can consist of millions to billions of cells and vertices. Hence the coordinates of the vertices and the topology of the cells, are considered heavy data and therefore stored in a separate file in the HDF5 format.

2.1 Shortcomings of the old Format

To understand the need for a new format and justify the effort of development and inclusion into SeisSol, it is necessary to evaluate the shortcomings of the old format.

There are multiple ways to save information about a mesh and restore it later on. The most obvious solution is, to save all known and required data in the file. With this data the information can be restored without any calculations on part of the reader. Therefore it is a fast way to read data but results in huge file sizes. The old format, which is based on NetCDF-4. Data is saved within a multidimensional array, to separate the partitions in the file.

NetCDF-4, which is a wrapper of HDF5, enforces for parallel reading, that multidimensional arrays are not ragged [8]. Therefore the array holding all partitions is fixed in both dimensions. If partitions are smaller than others, this leaves part of their space of the array empty. Because distributed simulations normally solve the same amount of elements per node, what results in equally sized partitions, this was no major problem at the time of creation of the old format. Since local timestepping was implemented in

SeisSol by Breuer, some elements are more often touched than others [9]. Therefore the amount of work for each node, does not correlate to the elements per partition anymore. Hence the partitions need to be generated with significantly different sizes, to balance the work. This results in wasted space within the multidimensional array and increases file size unnecessarily.

Additionally to the large file sizes imposed by the old format, it lacks flexibility. Using applications are for example bound to the tetrahedron as topology type. Furthermore the additional data which can be saved in the format is limited to values currently needed by SeisSol.

Because the old format was custom-made for SeisSol, compatibility with other applications was no major goal. Therefore the format is not viewable by visualization tools.

2.2 Why XDMF and HDF5?

One of the expectations to the new format is the viewability by visualization software such as ParaView and VisIt. Since it would be an exaggeration to create new plugins for each application, the necessity to base PXDMF on already supported formats emerged. To support various simulation applications, PXDMF is not allowed to restrict the data written to the file. Applications needing additional information to the unstructured mesh, should be able to attach this data to the file and visualize it with the mesh together. Therefore the best solution is to separate the description of the data from the data itself and add references between both.

ADIOS is a format, supporting references to data files, which is widely supported by visualization tools. Paraview shows the strictest limitations, and only reads files, which have been written with the vtkADIOSWriter included in the VTK project [10]. Documentation for this writer is sparse and it is not guaranteed, that the written file can be visualized by other applications, what makes ADIOS inapplicable as basis for PXDMF.

VTK provides its own format, for unstructured grids, which stores its data in binary files while providing the structure of the data in an XML file. This format can be read in parallel mode and attach additional user data. While being a good match for the needs of PXDMF, the Reader and Writer classes depend on VTK, which is not preinstalled on many clusters and has to be built separately.

Kitware, the publisher of VTK, developes another format, named XDMF. This format is independent from VTK and provides all functionality required for PXDMF. As described above, XDMF saves the data structure in XML and references to external files, which hold the heavy data. XDMF is supported by ParaView and VisIt and both applications

can visualize additional data included in the format. Therefore XDMF was chosen as basis for PXDMF.

As the data structure is known to the reader of PXDMF, it can ignore the XML file of XDMF and only read the heavy data. Still this file is needed for visualization applications.

While XDMF fixes the way light data is saved, heavy data can still be saved in HDF5 or binary files. HDF5 has its own library providing many functionalities, including parallel reading and writing with MPI [11]. Which reduces development effort in comparison to using a binary file extremely. Hence HDF5 and its library are used by PXDMF as basis. HDF5 stores its data in a hierarchical structure of datasets, which is basically comparable to a linux file system, with groups as folders and datasets as files, as shown in Figure 2.1.

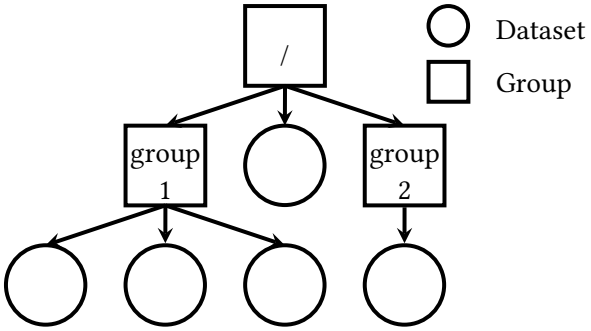


Figure 2.1: HDF5 stores data in a hierarchical structure of groups and datasets comparable to a file system of linux.

2.3 Structure

PXDMF is optimized for fast access when saving unstructured meshes with multiple partitions, while keeping file sizes small. Therefore it saves only five informations about the mesh, from which the mesh can be reconstructed in its whole. Those five informations are:

- globally unique id of each vertex
- coordinates of each vertex
- partitions a vertex is shared between
- the mesh connectivity
- topology type of the cells in the mesh

Based on HDF5 all data in PXDMF is structured in datasets, which are all located in the root group of the file. Each above information about the mesh is stored in its own

dataset, sorted by partition. The structure of a HDF5 file in the PXDMF format is shown in Figure 2.2.

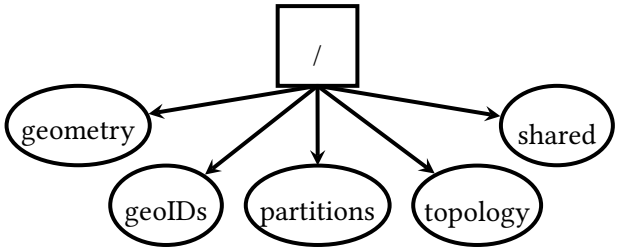


Figure 2.2: Structure of a HDF5 file in the PXDMF format.

Each node has to read only one block of continuous data per dataset. To enable them to find their blocks of data, PXDMF provides an additional dataset with offsets and sizes for each block. As shown in Figure 2.3, a node can aquire the start indices and sizes of its blocks by reading one row. Those informations are sufficient to access each block.

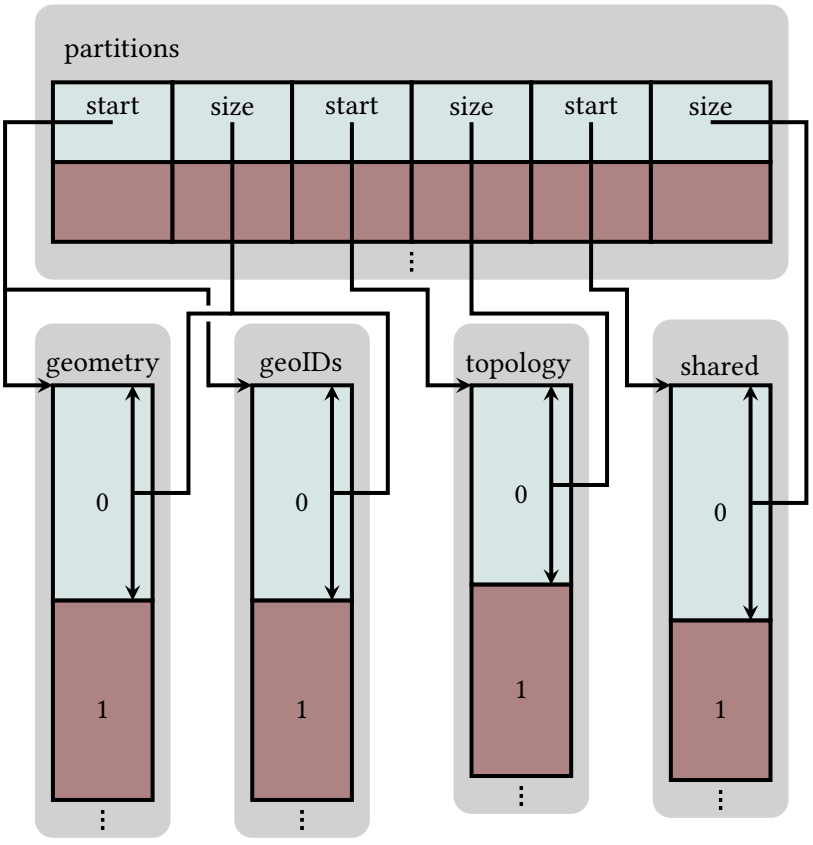


Figure 2.3: The dataset `partitions` holds the offsets and sizes of each block, a node has to read, in one row of the table.

The coordinates of vertices are stored in the dataset `geometry`. The first dimension of the dataset holds all vertices, while the second dimension is a array of the X, Y and Z coordinates. As described previously, vertices shared between multiple partitions are duplicated over their corresponding blocks of data. This is a slight overhead in terms of file size, that is needed for performance and can not be fully avoided. It would be possible to overlap the partitions, to make this duplication superfluous. This would need additional sorting of the partitions while writing the file format. While needing more time in the writing process, this would only have limited success in terms of minimizing the file size, since not all partitions sharing vertices could be written next to each other. Therefore it was chosen to minimize the generation time, while accepting the slightly larger file size.

The globally unique ids of the vertices need to be saved separately from the coordinates, because ParaView does currently not support hyperslabs in geometry datasets. Additionally the ids are integer, while the coordinates are of type double. The seperation allows storing both types without casting, which might be insecure, due to the inaccuracy of doubles for high values and integer for small values.

To optimize the communication between the nodes, while reconstructing the mesh, PXDMF saves the different partitions for each shared vertex. While the mesh is rebuilt, the nodes need to exchange data to establish globally unique ids for the shared edges and faces. With the information about the partitions, vertices are shared between, nodes can use point-to-point communication in contrast to broadcasts over all nodes. Since the partitions a vertex is shared with can fluctuate in number, a multidimensional dataset would have arrays in the second dimension with different lengths. HDF5 provides a special variable size datatype for exactly this problem, which is unfortunately not supported by MPI and thereby the parallel API PHDF5 [8], used by PXDMF. Saving the data in a multidimensional array with fixed length in both dimensions, would result in large quantities of unused space, since one dimension would depend on the maximum of partitions, a vertex is shared between. Hence all vertices with less partitions would leave empty space. Therefore a different solution had to be found, which neither uses a multidimensional dataset with fixed dimensions nor datatypes with variable length.

PXDMF uses a modified contiguous ragged array representation [12], to store the shared vertices with their partitions. Instead of storing the length of each block outside the array, it is stored in front of the block. Therefore a reader knows the blocks length, before accessing it. As all stored values only need to be read in order of appearance, no positions but only the data has to be stored. Hence data is saved, in a logical data structure as seen in Figure 2.4. Those structures are saved one after another in the array, wich results in a one-dimensional dataset. In the list of sharing partitions, the partition whithin whose block this structure is stored, can be omitted.

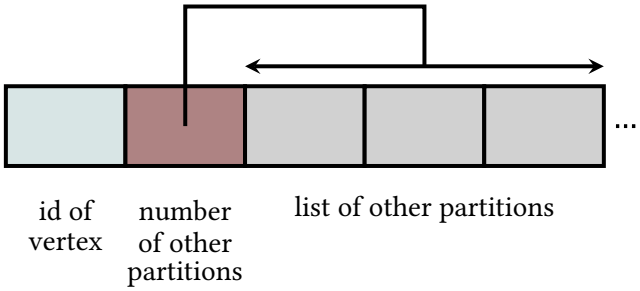


Figure 2.4: Logical data structure of a shared vertex

Finally the last two informations about the mesh, needing to be saved in the format, belong to the connectivity of the mesh and are saved in the dataset `topology`. Since cells can not be shared between partitions, the first dimension of the dataset is the number of cells within the mesh. Each cell is described by giving a list of indices to the vertices, which describe the corners of the topology. But different types of topology need more or less vertices to be fixed. Therefore the second dimension of the dataset, holding those vertices, depends on the stored topology, as shown in Figure 2.5.

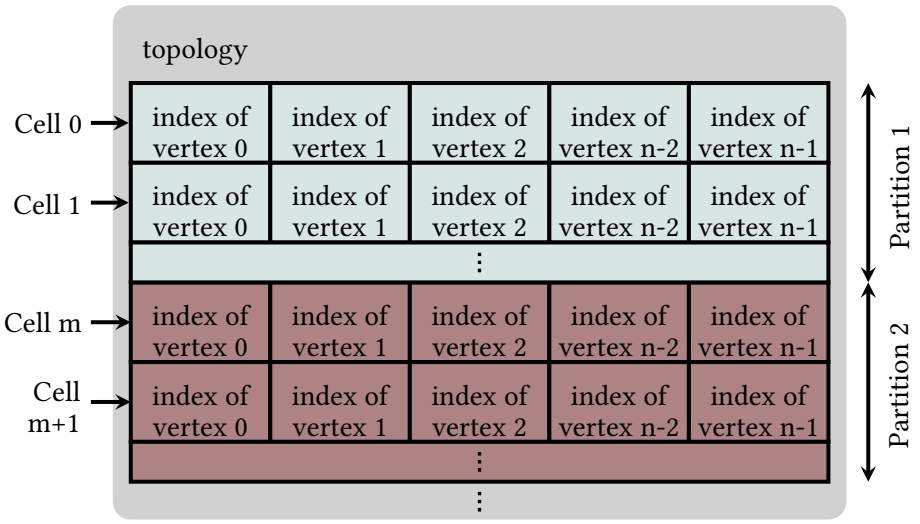


Figure 2.5: Dataset topology with n vertices per cell

Additionally to the different number of corners each topology has, it is also important in which order they construct the cell. The order decides which coordinates a face of the topology has and if wrongly interpreted by visualization tools, the rendered grid will look distorted. Therefore a list of the topology types and order of their corners is provided in Section 3.1. PXDMF supports most simple two-dimensional and three-dimensional topologies but enforces, that a mesh can only hold one topology type. The

following topology types are supported by PXDMF:

- Polygon
- Triangle
- Quadrilateral
- Tetrahedron
- Pyramid
- Wedge
- Hexahedron

A special type of topology is the polygon, since it has no fixed number of vertices. Therefore this number can be determined by the user but has to be constant over all cells of the mesh.

In the following a breakdown of the XDMF file is given, to finish the description of the format. The file looks mostly the same for each output of PXDMF, with little changes to some variables, linking the HDF5 file.

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
3  <Xdmf Version="2.0">
4      <Domain>
5          <Grid>
6              <Topology TopologyType="tetrahedron" NumberOfElements="
                   ↪ 166040">
7                  <DataItem NumberType="UInt" Precision="16" Format="HDF"
                       ↪ Dimensions="166040 4">
8                      test.h5:/tetrahedron
9                  </DataItem>
10             </Topology>
11             <Geometry GeometryType="XYZ" NumberOfElements="30030">
12                 <DataItem NumberType="Float" Precision="8" Format="HDF"
                       ↪ Dimensions="30030 3">
13                     test.h5:/geometry
14                 </DataItem>
15             </Geometry>
16         </Grid>
17     </Domain>
18 </Xdmf>
```

Listing 2.1: XDMF file of PXDMF (test file distributed with PXDMF)

Note the version number of XDMF 3. Kitware, the developer of ParaView and XDMF, chose to internally differ from the official versioning [13]. Hence the internal version of XDMF 3 is 2.0.

Within the XDMF file, as shown in Listing 2.1, the elements `Topology` and `Geometry` are of great importance. They give the visualization tool informations on where to find the data and how to read it. Both contain an element named `DataItem`. This item references the heavy data file and points to a dataset within it. In the various attributes of the elements in XDMF are informations about the datatypes and dimensions of the datasets. With those it is possible, to read the data and visualize the meshes.

PXDMF allows adding additional data to files, which is described in Chapter 3.3.2. This data is saved in separate datasets, either within the heavy data file holding the mesh or externally in a separate file. The library of PXDMF adds an element, as shown below, to the XDMF file for each additional dataset. This enables users to view the data in visualization tools.

```
1 <Attribute Name="cellData" Center="Cell">
2     <DataItem NumberType="UInt" Precision="8" Format="HDF" Dimensions="
      ↪ 166040 1">cellData.h5:/cellData</DataItem>
3 </Attribute>
```

Listing 2.2: element for additional data within a XDMF file of PXDMF

Chapter 3

Library

PXDMF provides a library for fast parallel access to unstructured partitioned meshes. The library is written in C++ Standard 98, but provides a preprocessor macro for improved handling of maps with the Boost library. This macro exchanges regular maps of C++ with unordered maps from Boost. If the library is compiled with C++11 or higher, the unordered maps included in the C++ specification are used, even if the macro is set.

PXDMF supports writing to and reading from files, as well as attaching additional data to a written mesh. The additional data is either attached to the cells or the vertices of a mesh and can be read during initialization of the mesh, if the library is instructed to do so. The data will be attached to the internal data structure and can be accessed thereby. This chapter provides an overview of the data structure, through which a mesh can be accessed, the functions provided to users and the algorithms used internally.

3.1 Data Structure

The data structure provided by PXDMF implements the general topology-based mesh data structure of Beall and Shephard [14]. Therefore time needed for traversal in the mesh over adjacent topological entities is independent from the number of entities in the mesh. For fast access to all elements of a mesh, PXDMF provides adjacencies for entities, either one dimension upward or downward. Thereby the traversals shown in Figure 3.1 are possible.

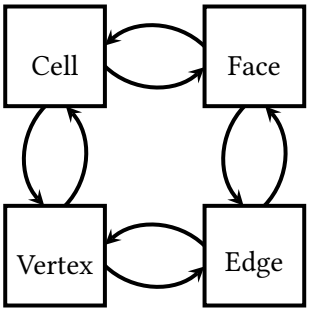


Figure 3.1: Supported traversal graph over all entity types provided by PXDMF

In PXDMF all elements, no matter what entity type, are of the data type `Element`. Depending on their entity type they are maintained in one of four arrays, named by the entity type they store. Each element has two IDs, a globally unique id, saved within the data structure as `globalID`, and a local id, implicated by the position in the array. To efficiently search for an element by its global id, maps are provided, which match each global id to the local id of an element.

In the library dimensional traversals are done by either querying the data members `upperElements` and `lowerElements`, which provide the local id of adjacent elements or by calling the member functions `traverseUp()` and `traverseDown()`. The functions return pointers to the corresponding adjacent elements and are described further below. All traversal functions and data members only provide elements of the current partition. Therefore special attention has to be paid to elements, shared between partitions. They are marked by a flag and have corresponding elements in each sharing partition. The partition and local id of each such element in a neighbouring partition is stored within an array.

Further each element stores the coordinates of its mean center and additionally its entity type, which is either `CELL`, `FACE`, `EDGE` or `VERTEX`. If an element is of entity type `CELL`, the topology is provided in the data member `topology`. For other entity types the topology is undefined.

data member	type	purpose
`lowerElements`	std::vector<unsigned long>	stores local ids of all adjacent elements one dimension downward
`upperElements`	std::vector<unsigned long>	stores local ids of all adjacent elements one dimension upward
`globalID`	unsigned long	globally unique id of the element
`x, y, z`	float	coordinates of the mean center of the element
`elementType`	ElementType	enumerated topology entity this element represents • possible values: – `CELL` – `FACE` – `EDGE` – `VERTEX`
`topology`	Topology	enumerated topology type this element represents • required if elementType is `CELL` • possible values: – `POLYGON` – `TRIANGLE` – `QUADRILATERAL` – `TETRAHEDRON` – `PYRAMID` – `WEDGE` – `HEXAHEDRON`
`isShared`	unsigned int	whether element is shared over multiple partitions or not
`neighbours`	std::vector<int>	ids of the partitions this element is shared with
`neighbourLocalIDs`	std::vector<unsigned long>	local ids of the corresponding element in the sharing partitions

Table 3.1: Public data members with type and purpose of the type Element

Additionally the data type Element provides a constructor, which sets essential values at creation time. Calling the constructor, without a topology type, will result in an exception, when the created topology entity is a cell. If the topology is provided for vertices, edges or faces, improved allocation of internal arrays is acchieved. The identity of the constructor is shown in Table 3.2.

```
Element(std::vector<Element> *upwardArray,
        std::vector<Element> *upwardArray,
        unsigned long globalID,
        ElementType dimension,
        Topology topology = UNSET)
```

Creates an element for a new cell, face, edge, vertex. If the topology entity is not set to `CELL`, the topology type is set but can be ignored.

variable	type	purpose
`upwardElements`	std::<vector<Element>*	pointer to the array storing all elements one dimension upward
`downwardElements`	std::<vector<Element>*	pointer to the array storing all elements one dimension downward
`globalID`	unsigned long	globally unique id of the new element
`dimension`	ElementType	enumerated topology entity of the new element possible values: • `CELL` • `FACE` • `EDGE` • `VERTEX`
`topology`	Topology	enumerated topology type the new element represents possible values: • `POLYGON` • `TRIANGLE` • `QUADRILATERAL` • `TETRAHEDRON` • `PYRAMID` • `WEDGE` • `HEXAHEDRON`

Table 3.2: Constructor of the type Element

To simplify traversal through adjacent elements, the type Element provides three functions, additionally to the arrays described previously. The functions `traverseUp()` and `traverseDown()` allow dimensinal traversals. They return pointers to elements instead of their local ids, as the data members `lowerElements` and `upperElements` are doing. The function `removeSelf(...)` removes the calling element from an array of pointers to elements.

`std::vector<Element *> traverseUp()`
Return pointers to all adjacent elements one dimension upward.

Table 3.3: Function for traversing one dimension upward

`std::vector<Element *> traverseDown()`
Return pointers to all adjacent elements one dimension downward.

Table 3.4: Function for traversing one dimension downward

`std::vector<Element *> removeSelf(std::vector<Element *> vector)`
Remove the calling element from the array `vector` and return the other elements.

variable	type	purpose
`vector`	std::<vector<Element *>	array of pointers to elements, from which the calling element should be removed

Table 3.5: Function to remove the calling element from an array

This function is especially advantageous when traversing multiple times in different directions. Hence searching for neighbours of a cell, sharing a face, is done in four lines of code, independent from the number of neighbors, as seen in Listing 3.1.

```
1 std::vector<Element *> neighbors;
2 std::vector<Element *> lElements = cell.traverseDown();
3 for (auto it = lElements.begin(); it != lElements.end(); ++it)
4   neighbors.push_back(cell.removeSelf(it->traverseUp()));
```

Listing 3.1: Searching for all neighbours of an element

If additional data[1] is read while initializing the mesh, it is added by PXDMF to the corresponding elements in the mesh. When loading such data, the user has to provide the name of the dataset. This name is later used, to access this data via the elements. For this purpose the type Element has the member function `getAttachedDataByName(...)`, which is detailed in table 3.6

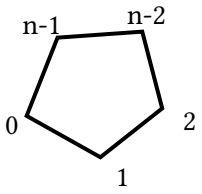
[1] See Section 3.3.2 for further information on adding additional data.

`std::vector<T> *getAttachedDataByName(std::string name)`
Get a pointer to the data attached to this element with the given name.

variable	type	purpose
`name`	std::string	name of the dataset from which the data was loaded

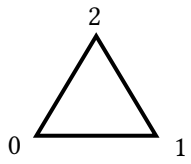
Table 3.6: Function to aquire previously loaded data from an element

In PXDMF it is possible to access specific topological entities while traversing the mesh. Therefor the faces, edges and vertices of each topology type are specifically ordered. The following figures provide an overview of all supported types of topology. The numbers specifying a face, edge or vertex are the corresponding positions in the arrays `upperElements` and `lowerElements` and the ones returned by all functions providing dimensional traversal. The ordering in PXDMF is based on the GAMBIT format [15], with modifications to ensure compatibility with ParaView, which enforces the vertex ordering [16].



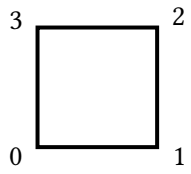
Edge	Vertices	Face	Vertices
0	0,1	0	0,1,...,n-2,n-1
1	1,2		
2	2,n-1		
n-2	n-1,n-2		
n-1	n-2,0		

Table 3.7: Polygon with n vertices



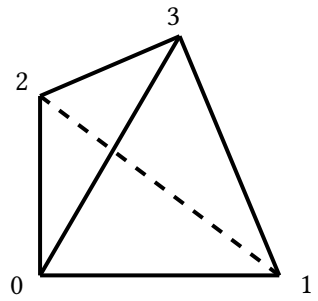
Edge	Vertices	Face	Vertices
0	0,1	0	0,1,2
1	1,2		
2	2,0		

Table 3.8: Triangle



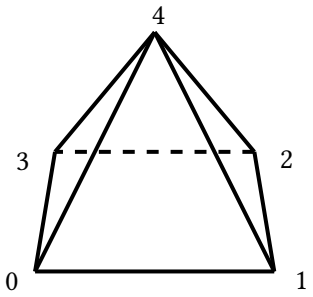
Edge	Vertices	Face	Vertices
0	0,1	0	0,1,2,3
1	1,2		
2	2,3		
3	3,0		

Table 3.9: Quadrilateral



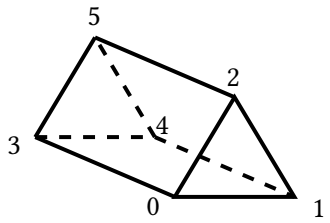
Edge	Vertices	Face	Vertices
0	0,1	0	0,1,3
1	1,2	1	1,2,3
2	2,0	2	2,0,3
3	0,3	3	0,1,2
4	1,3		
5	2,3		

Table 3.10: Tetrahedron



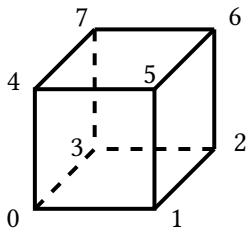
Edge	Vertices	Face	Vertices
0	0,1	0	0,1,2,3
1	1,2	1	0,4,1
2	2,3	2	1,4,2
3	3,0	3	2,4,3
4	0,4	4	3,4,0
5	1,4		
6	2,4		
7	3,4		

Table 3.11: Pyramid



Edge	Vertices	Face	Vertices
0	0,1	0	0,2,5,3
1	1,2	1	2,1,4,5
2	2,0	2	1,0,3,4
3	3,4	3	0,1,2
4	4,5	4	3,4,5
5	5,3		
6	0,3		
7	1,4		
8	2,5		

Table 3.12: Wedge



Edge	Vertices	Face	Vertices
0	0,1	0	0,4,5,1
1	1,2	1	1,5,6,2
2	2,3	2	2,6,7,3
3	3,0	3	3,7,4,0
4	4,5	4	0,1,2,3
5	5,6	5	4,5,6,7
6	6,7		
7	7,4		
8	0,4		
9	1,5		
10	2,6		
11	3,7		

Table 3.13: Hexahedron

3.2 Reading Format

Each mesh, PXDMF reads, is divided into a number of partitions. When reading meshes, the number of ranks and partitions need to be the same. While accomplishing best performance on a homogenous cluster, where each node reads its own data, PXDMF has builtin support for heterogenous clusters, where only some nodes have access to the file. To support such clusters, nodes are grouped together, with the first node per group reading and distributing data. This node iterates over all partitions of the group in reverse order and delivers the data to the corresponding ranks. The last partition read, is processed by the reading node itself.

Since the file format does only contain essential information, the library needs to recalculate the mesh. For this, PXDMF builds the adjacency graph shown in Figure 3.1 step by step. In the beginning it iterates over all cells in the partition and creates corresponding elements in its internal data structure. After creating a new cell, PXDMF constructs all vertices, this cell references and links them, according to the structure described in Section 3.1. Special attention needs to be paid to vertices already created,

since duplicates are not allowed, or traversing in the mesh would result in undefined behaviour. Therefore PXDMF remembers already created vertices and references those, in case of multiple cells depending on them. With the order of vertices, declared for each topology type, edges and faces between the vertices can be created. For those elements the same restriction, of having no duplicates, is enforced. This is done by comparing adjacent elements, one dimension upward, of all lower elements of the new one.If there exists an intersection, the resulting element is chosen and linked, according to the adjacency graph.

While creating edges, it can be determined, if the edge is shared between partitions. This is done by querying the sharing state of its vertices. If both vertices are shared, the edge might also be shared. While all shared edges are found with this algorithm, the following exception exists.

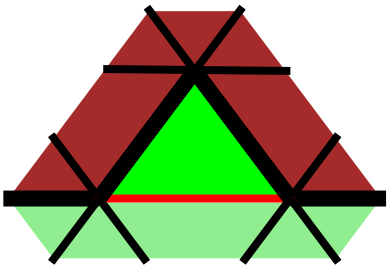


Figure 3.2: Special case to consider when shared edges are deduced by querying the sharing state of their vertices. Broad lines are shared edges on partition borders and the red line is wrongly marked as shared.

In Figure 3.2 all three vertices of the triangle are shared between partitions, while only two of the three edges are shared. PXDMF uses the previously described algorithm to find shared edges. Hence it needs to filter those edges, which were wrongly marked as shared. This is done while synchronizing the global ids of elements, and will be explained together with that algorithm.

Setting the shared state of the faces is delayed until the shared state of all edges was validated, to ensure no spreading of errors. Shared faces are deduced by the same algorithm as edges, but with the edges as the lower elements being queried. While the same exception occures, as with the edges, in terms of the faces, this can be mostly rectified within the partition and therefore minimal communication. As described in Section 3.1, elements reference only adjacent elements in the same partition. Therefore shared faces can only have one adjacent cell, while non-shared faces have two. Thus faces with only one adjacent cell and all their edges shared, are either shared or at a global boundary. Hence cells at a global boundary with all edges shared need to be filtered.

To create unique elements within the data structure of PXDMF it is also necessary to

globally assign unique ids. Cells have their id indirectly saved within the file, as the index by which the row of the cell is accessible. While aquiring the global id is easily done for vertices and cells, where it is saved within the file, it is not as straightforward for edges and faces. Edges and faces are completely omitted from the file format, which also means, that there are no ids directly or indirectly saved. Therefore each element is assigned the position within the holding array as its global id with an additional offset of one. The offset is necessary to ensure a globally unused id, which is needed by some algorithms. This genereates locally unique ids but leads to collisions within the global id space. Hence the ids need to be modified to ensure differentness. This is done by counting elements of all lower partitions[2] and adding this as offset to the ids of all elements of the partition. While all collisions within the global id space are now resolved, shared elements need to have the same id over all partitions. Therefore an exchange between the nodes is necessary, to align the global ids of those elements. Within this process, it is possible to rectify the wrongly shared elements of the special case, explained previously.

In PXDMF the lowest partition, sharing an element, assignes the elements global id to all corresponding elements within other partitions. To parallelize communication, each node sends all its shared elements, of the same topological entity, in one message per neighbouring partition. As format within the messages, the modified jagged array structure of Figure 2.4 is used. For each shared element the sent data consists of the global id, if the sender is the lowest partition sharing this element, or zero. Attached to the id, a list of the global ids of all adjacent elements one dimension downward is sent.

For each received message, a partition modifies the elements accordingly, by setting neighbours and, if given, the global id. Additionally the received elements of a message are compared with the elements sent to the corresponding partition. If more elements were sent to a partition than were received, those elements need to be unmarked as shared, since they are mistakenly marked, which solves the special case previously described. Removing the shared state of wrongly shared elements while exchanging ids requires minimal communication. Since multiple steps are condensed into few communciations, an exception can occure under some circumstances. For this to happen, the lowest sharing partition of an element has to be wrongly marked as such, and therefore unavailable to distribute the global id of the element, since it is unknown to the partition. Hence nobody distributes the id and after all communication is finished, the element has different ids over all sharing partitions. To amend this situation, another communication between all sharing partitions is necessary, where the now lowest partition distributes the id.

Since this scenario could also implicate, that a partition is sharing only that one face with another partition, and therefore sends a message but will receive none in return, PXDMF needs to send messages between all partitions, even if there are no shared faces.

[2] The id of a partition is determined by its row index in the partition dataset within the file.

This stems from the used communication library MPI [17], where all sent messages need to be received (to free buffers) [18] and waiting for a messsage, which is never sent, takes unnecessary time and is therefore inefficient. After assigning all global ids correctly, they can be mapped to the local ids and stored in the corresponding maps of the library, what finishes the initialization of the internal data structure.

The function declaration, implementing all this functionality, is:

```
void readFile(std::string fileName,
              std::vector<std::string> attachedDataNames,
              int groupSize = 1,
              MPI_Comm comm = MPI_COMM_WORLD,
              MPI_Info info = MPI_INFO_NULL)
```

Reads a file of the PXDMF format and initializes the internal data structure with the mesh. Can read additional data, and attach it to corresponding elements of the mesh.

variable	type	purpose
`fileName`	std::string	path to the file containing the mesh
`attachedDataNames`	std::<vector<std::string>	vector with identifiers for additional data identifiers are of type: • `file:dataset` if dataset is not an external file • `dataset` if dataset is in the same file as the mesh
`groupSize`	int	group size for heterogenous clusters (first node per group reads file)
`comm`	MPI_Comm	communicator to use for reading the file
`info`	MPI_Info	MPI information to use for opening HDF5 file

Table 3.14: Function to read a file in the PXDMF format

3.3 Writing Format

PXDMF supports distributed writing of unstructured meshes, where each node only holds part of the whole mesh. Additionally the number of partitions of the mesh does not need to be equal to the number of nodes writing, but needs to be dividable by it. To write a mesh, each node needs to provide the connectivity of the cells in its partitions and a set of the coordinates of all vertices contained in the mesh. The vertices provided by a node, do not need to be the vertices required by the partitions of this node. Therefore PXDMF distributes all vertices to the nodes, requiring them. For this task PXDMF includes an algorithm, which is a modified version of the Broker algorithm implemented by Rettenberger [19].

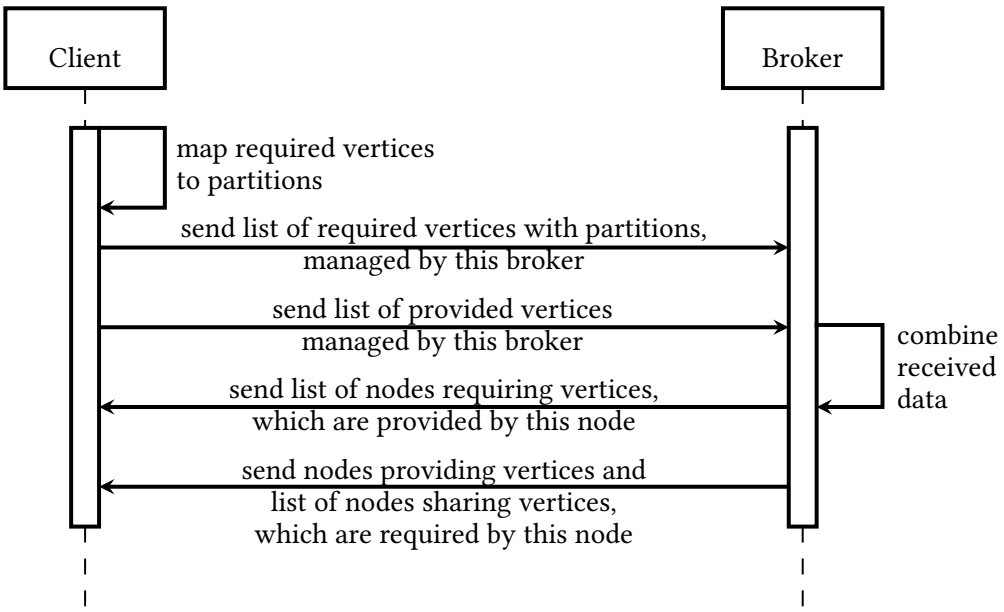


Figure 3.3: Initialization phase of the broker algorithm between a client and a borker

While using the broker algorithm, as shown in Figure 3.3, each node is a broker and client at the same time. In contrast to the algorithm created by Rettenberger, PXDMF separates the intialization phase from the phase, where the data is transmitted. This allows for multiple data transmissions on the same communication structure.

For the broker algorithm all vertices of a mesh are separated into disjoint sets, from which each broker manages one. For this set all nodes send the ids of the vertices to the corresponding broker, from which they either store the data or from which they need the data. This information is combined by the broker, to establish a communication matrix. After establishing this matrix, a list of all nodes requiring a vertex is sent to the node providing the data of this vertex. And for each vertex, managed by the broker, the id of the node providing the data and a list of all partitions including this vertex, is sent

to each node requiring the vertex.

After this initialization phase, each node has the information needed to build the dataset *sharing* when writing a file, and can transmit data, attached to vertices, with point-to-point communication between nodes.

3.3.1 Writing Mesh

To write a mesh with PXDMF, the library provides a function, with the indentity, shown in Table 3.15. Within this function, the broker algorithm is used, to exchange the coordinates of the vertices and thereby sort the vertices into single blocks per partition.

If the file referenced in `fileName` already exists, PXDMF overwrites it with the new mesh. The vertices are presented to the function in two parts. In `rankVertices` the coordinates of each vertex are saved, while the ids of those vertices, saved in `rankVertexIDs`, have to be stored in the same order as the coordinates. Additionally the connectivity of the cells has to be provided in the vector `rankCells`, which is built like the topology dataset, as shown in Figure 2.5, except that it is a one-dimensional array with the rows concatenated. The number of vertices per cell is implicated by the topology provided in `pTopo`, or, if the topology is of type POLYGON, by the user in `pNodeVertices`. As described previously, only the cells for wich the connectivity is provided to a node, need to be the ones, contained in the partitions of the node. Those partitions are given in the array `rankPartitionSizes`, which holds the number of cells to read from the array `rankCells` per partition. Before writing the datasets, the ids given in `rankCells` need to be exchanged with the indices of the rows, where the corresponding vertices are later on saved in the file. Only by exchanging this information, visualization applications can view the mesh. Finally the data is written to a HDF5 file with the same file name as the XDMF file but the ending `.h5`. Additionally a XDMF file as shown in Listing 2.1 is generated with the light information of the unstructured mesh. The function `writeFile` returns the broker created for exchanging vertices between partitions. This broker can be reused to attach data to the vertices of the grid.

```
static Broker &writeFile(std::string fileName,
              std::vector<double> *rankVertices,
              std::vector<unsigned long> *rankVertexIDs,
              std::vector<unsigned long> *rankCells,
              std::vector<unsigned long> rankPartitionSizes,
              Topology pTopo,
              unsigned long pCellVertices = 0,
              MPI_Comm comm = MPI_COMM_WORLD,
              MPI_Info info = MPI_INFO_NULL)
```

Writes a mesh to a file with the PXDMF format.

Returns: The broker used to redistribute the vertices

variable	type	purpose
`fileName`	std::string	path to xdmf file, the mesh should be written to
`rankVertices`	std::<vector<double> *	pointer to an array holding all coordinates of the vertices
`rankVertexIDs`	std::<vector<unsigned long> *	pointer to an array holding all ids of the vertices
`rankCells`	std::<vector<unsigned long> *	pointer to an array holding all ids of the vertices of each cell
`rankCells`	std::<vector<unsigned long>	array holding the number of cells to read per partition
`pTopo`	Topology	pointer to an array holding all ids of the vertices of each cell possible values: • `POLYGON` • `TRIANGLE` • `QUADRILATERAL` • `TETRAHEDRON` • `PYRAMID` • `WEDGE` • `HEXAHEDRON`
`pCellVertices`	unsigned long	the number of vertices per cell can be omitted if cells are not of topology type POLYGON
`comm`	MPI_Comm	MPI communicator to use for reading the file
`info`	MPI_Info	MPI information to use for creating HDF5 file

Table 3.15: Function to write a file in the PXDMF format

3.3.2 Writing additional Data

PXDMF allows attaching of additional data, which can be viewed by visualization software and read by this library. Such data needs to be attached to a entity type of a mesh. PXDMF supports attaching to either cells or vertices and users must provide data for each element of this type within the mesh. If the data is not provided consistently, the behaviour is undefined. Additionally the order of the data is important, as it must correspond to the order of the elements in the file, the data should be attached to. For a user of the library this means, that the data has to be provided in the same order as the elements were given to the function `writeFile`.

For data attached to vertices the same rules as for the vertices are applied, which are mentioned in Section 3.3.1. Therefore PXDMF reuses the broker provided by the function `writeFile` to redistribute the data.

Further attached data can either be stored in the same heavy data file as the mesh or in its own file. Additionally this data can be loaded at initialization time selectively, to ensure minimal memory footprintby only loading required data.

It is possible to attach multiple values of the same data to one element. Therefor the data for each element has to be concatenated in the array `pData`, with the variable `stride` giving the number of values per element, as seen in Figure 3.3.2.

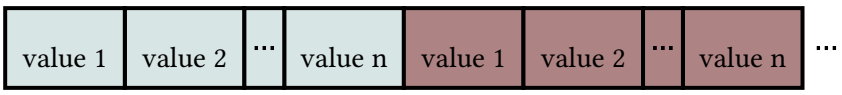


Figure 3.4: Data in array `pData` with stride `n`

The variable `dataName` containes the name of the dataset, where the data is written. When reading the mesh, the attached data can be accessed via this name. Attached data can be one of several datatypes. While all data saved in one dataset has to be of the same type, it is possible to save multiple datasets with different types. The supported datatypes are listed in Table 3.16.

If the data should be saved in its own HDF5 file, the path to this file has to be provided in `extDataFileName`. If this variable is an empty string, the data will be attached to the heavy data file of the mesh. To link the data to the mesh for visualization software and to support data being stored directly with the mesh, the path to the xdmf file is required in the variable `xdmfFileName`.

```
template<typename T>
static void attachDataToFile(std::vector<T> *pData,
              DataType dataType,
              unsigned long stride,
              std::string dataName,
              AttachTo attachTo,
              std::string xdmfFileName,
              std::string extDataFileName = "",
              Broker *broker = NULL,
              MPI_Comm comm = MPI_COMM_WORLD,
              MPI_Info info = MPI_INFO_NULL)
```

Attaches additional data to the vertices or cells of a mesh.

variable	type	purpose
`pData`	std::vector<T>*	vector holding all data for the elements in the format shown in Figure 3.3.2
`dataType`	DataType	Data type of the values provided possible values: • `CHAR` • `UNSIGNED_CHAR` • `INT` • `SHORT` • `LONG` • `LONG_LONG` • `UNSIGNED_INT` • `UNSIGNED_SHORT` • `UNSIGNED_LONG` • `UNSIGNED_LONG_LONG` • `FLOAT` • `DOUBLE`
`stride`	unsigned long	number of data values to attach to each element
`dataName`	std::string	name of the dataset where the data should be stored
`attachTo`	AttachTo	the topology entity where the data should be attached to possible values: • `CELLS` • `VERTICES`
`xdmfFileName`	std::string	path to the XDMF file of the mesh

`extDataFileName`	std::string	path to the file where the data should be saved (or empty path if it should be saved with the mesh)
`broker`	Broker*	The broker returned by writeFile() or NULL if attaching data to cells
`comm`	MPI_Comm	MPI communicator to use for reading the file
`info`	MPI_Info	MPI information to use for creating HDF5 file

Table 3.16: Function to attach data to a file in the PXDMF format

Chapter 4

Tests

To evaluate PXDMF, we compare it to the old format used between preprocessing software and SeisSol. The main points of comparison are time needed for generation and initialization of the mesh. Additionally the file size depending on partitions and meshsize is assessed. All benchmarks, measuring initialization or generation times, are mean measurements of multiple runs.

The tests have been executed on the Linux Cluster [20] of the Leibniz Computing Centre. This cluster consists of several segments with different architectures and limits. For the tests the CoolMUC2 segment was used, which uses Intel Xeon E5-2690 v3 cpus with 28 cores and 64 GB of RAM per node. All tests used the MPI library the Intel MPI [21] in version 5.1. The Linux Cluster is a homogeneous cluster, where each node can access the file system. The file system, used for the tests, is a dedicated Network Attached Storage system, which offers 30 GB/s I/O-Performance with small and big files.

4.1 Generation

To generate comparable files in both formats, a modified version of PUMGen [4] was used. This version generates the same mesh in both formats, while measuring the time needed. Only the steps directly concerning the generation of the files were measured.

PXDMF stores vertices in its files, sorted by partitions. Therefore it requires a communication phase pervious to writing the data, where vertices are exchanged between nodes. This process, previously described as the broker algorithm, is distributed evenly between all nodes. As can be seen in Figure 4.1, PXDMF takes less time generating a file, the more often it is partitioned and distributed over nodes. This stems from the fact, that increasing the number of partitions leads to less vertices per rank, that need to be managed, thus amortizing the additional workload of the broker algorithm. If the partitions are further increased, after the cost of the broker is amortized, the effort

for each node, to communicate with each broker, increases exceptionally. Hence the generation time is increasing again.

Regarding mesh sizes, the scaling behaviour is mostly the same for both formats. A single exception is the point of even, where the libraries of both formats need the same amount of time, to generate the file. From this point ongoing PXDMF requires less time than the old format, to output the same data.

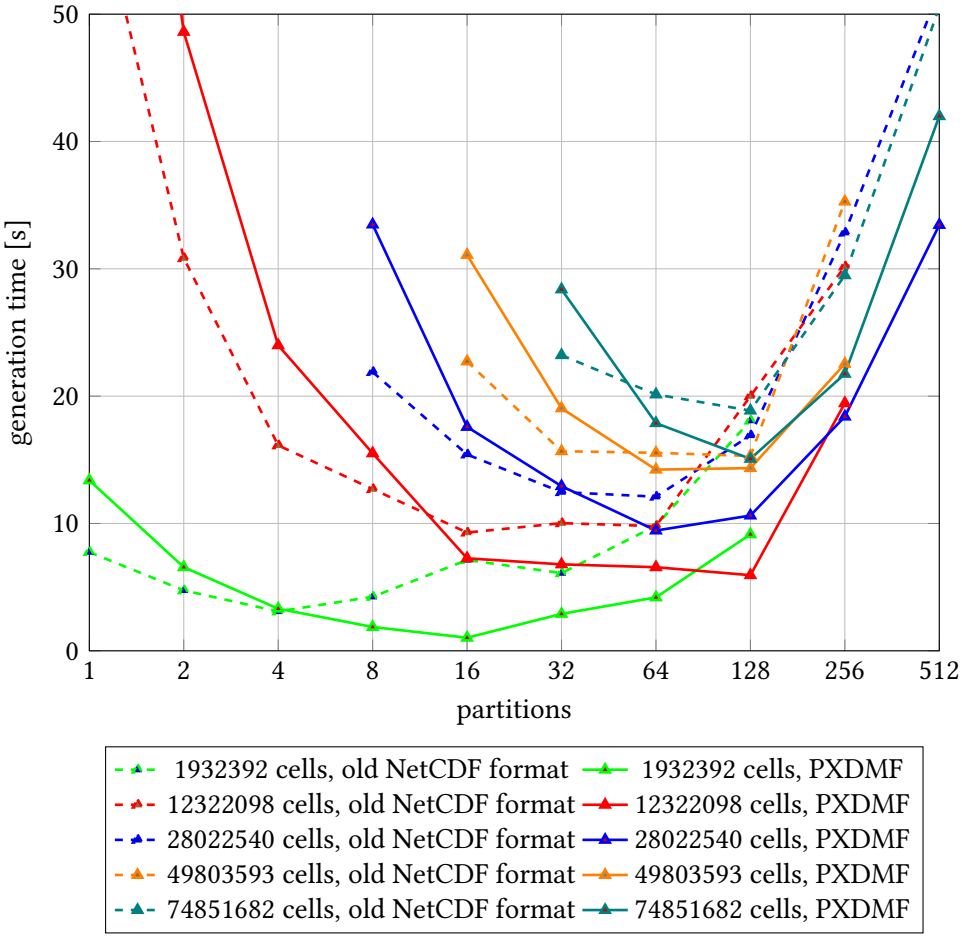


Figure 4.1: Generation time depending on the number of partitions.

4.2 Initialization

To benchmark initialization time, a small simulation software was implemented. This application reads both formats, and runs a simple simulation on the mesh. Before running the simulation, the mesh data is transfered to an internal data format, to ensure no favoritism of the internal structure of PXDMF. Still this internal data structure is initialized after PXDMF has initialized its own. Hence the workload of PXDMF is greater

than the one of the old format, which is directly generating the final data structure of the simulation software without additional steps in between.

This can also be seen in Figure 4.2, where the time consumption of PXDMF is consistently greater than the initialization time needed by the old format. Since PXDMF shows smaller growth rates when increasing the partitions, it is applicable as storing format for large unstructured meshes.

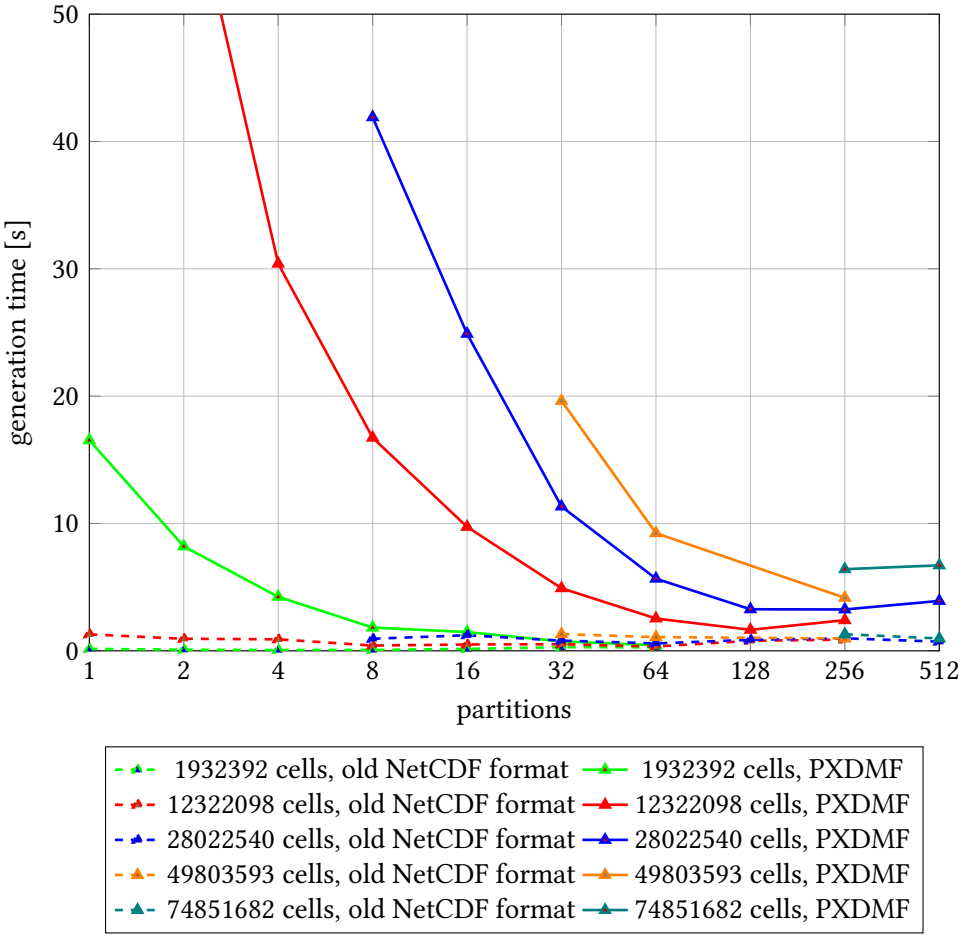


Figure 4.2: Generation time depending on the number of partitions.

4.3 File Size

One of the expectations to PXDMF is, that the format scales well concerning the file size and partitions. As can be seen in Figure 4.3, PXDMF scales nearly linear, with only a slight overhead, when the mesh is divided into more partitions. This additional data, stems from the slighlty more shared and therefore duplicated elements because of the partitioning.

Since the format is intended for simulations on large unstructured meshes, the scaling behaviour with many partitions is very important. As seen in Figure 4.3 the file size of PXDMF scales three times better, when compared to the old NetCDF format.

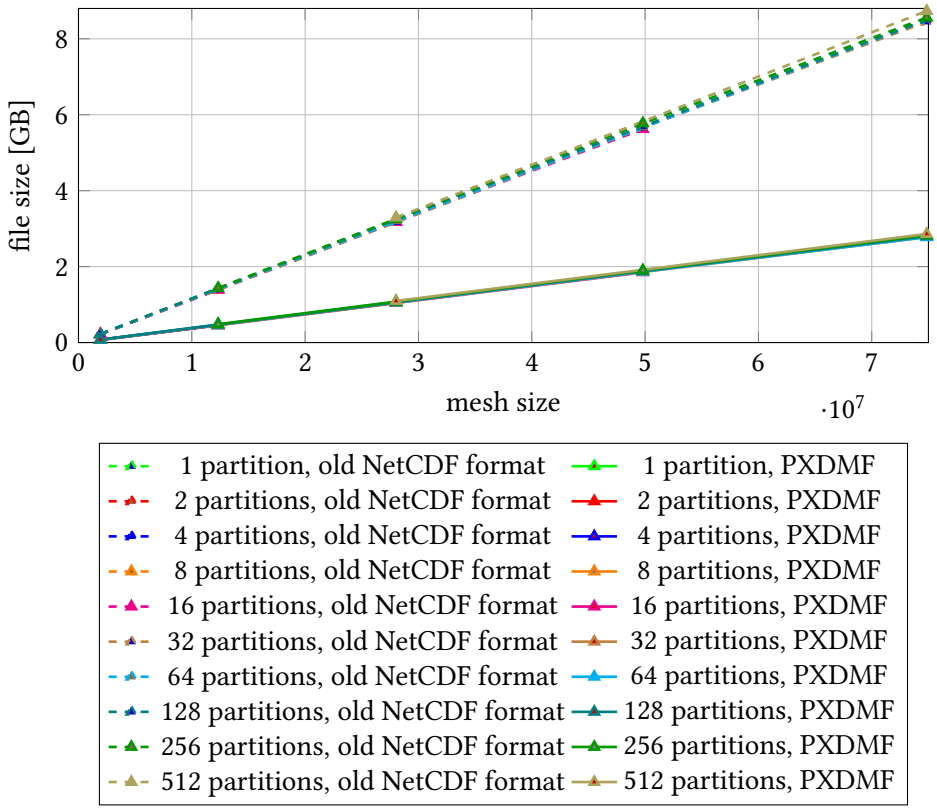


Figure 4.3: File size depending on the mesh size.

Chapter 5

Conclusion

As seen in the tests, PXDMF is a well scaling format for parallel unstructured meshes. The file size is kept very low, while ensuring good initialization times. With the linear scaling behaviour of the library for initialization scaling behaviour of using applications is not hindered.

PXDMF stores a read mesh in a traversable data structure, providing all information known about the mesh. A simulation application can use this data structure directly, to shorten the time needed for initialization or can generate its own structure with the data provided. A big advantage over the old format is the flexibility of PXDMF, which allows many different topology types and does not restrict additionally added information in amount or data type. Hence the data format supports many use cases. Finally PXDMF can be visualized with all additional data added to the mesh, allowing users to validate data by themselves, as well as showcasing meshes.

Bibliography

[1] S. Rettenberger and M. Bader, "Optimizing I/O for Petascale Seismic Simulations on Unstructured Meshes," in *2015 IEEE International Conference on Cluster Computing*, Sept 2015, pp. 314–317.

[2] "SeisSol," http://www.seissol.org/, accessed: 2017-02-22.

[3] "ParaView," http://www.paraview.org/, accessed: 2017-03-01.

[4] "PUMgen," https://github.com/TUM-I5/PUML, accessed: 2017-02-22.

[5] "XdmfWeb," http://www.xdmf.org/, accessed: 2017-03-01.

[6] "HDF Group - HDF5," https://support.hdfgroup.org/HDF5/, accessed: 2017-03-01.

[7] "VisIt," https://wci.llnl.gov/simulation/computer-codes/visit/, accessed: 2017-03-01.

[8] "Parallel HDF5," https://support.hdfgroup.org/HDF5/doc/TechNotes/VLTypes.html, accessed: 2017-03-09.

[9] A. N. Breuer, "High performance earthquake simulations," Dissertation, Technische Universität München, München, 2015.

[10] "[Paraview-developers] Paraview Adios," http://public.kitware.com/pipermail/paraview-developers/2015-July/003887.html, accessed: 2017-03-10.

[11] "Parallel HDF5," https://support.hdfgroup.org/HDF5/PHDF5/, accessed: 2017-03-01.

[12] B. Eaton *et al.*, "NetCDF Climate and Forecast (CF) Metadata Conventions," *CF-Conventions*, 2011, version 1.6.

[13] "[vtkusers] Fwd: XDMF and hyperslabs," http://www.vtk.org/pipermail/vtkusers/2015-August/092001.html, accessed: 2017-03-02.

[14] M. W. Beall and M. S. Shephard, "A general topology-based mesh data structure," *International Journal for Numerical Methods in Engineering*, vol. 40, no. 9, pp. 1573–1596, 1997.

[15] Fluent Inc., *GAMBIT NEUTRAL FILE FORMAT*, 2006.

[16] W. J. Schroeder, K. Martin, L. S. Avila, and C. C. Law, *The VTK user's guide*. Kitware, 2001.

[17] "MPI Forum," http://mpi-forum.org/, accessed: 2017-03-07.

[18] "MPI_Wait(3) man page (version 1.8.8)," https://open-mpi.org/doc/v1.8/man3/MPI_Wait.3.php, accessed: 2017-03-07.

[19] "SCOREC Core," https://github.com/SCOREC/core, accessed: 2017-03-07.

[20] "LRZ: Linux-Cluster," https://www.lrz.de/services/compute/linux-cluster, accessed: 2017-03-08.

[21] "Intel MPI Library," https://software.intel.com/en-us/intel-mpi-library, accessed: 2017-03-08.