HOPR HDF5 Curved Mesh Format

Florian Hindenlang, Thomas Bolemann

last modified: August 10, 2016

Institute for Aerodynamics and Gasdynamics,
University of Stuttgart

Contents

1 Introduction

1.1 Main idea behind the Mesh Format

The High Order Preprocessor (HOPR) is able to generate high order unstructured 3D meshes, including tetrahedra, pyramids, prisms and hexahedra. The HDF5 library (http://www.hdfgroup.org/) allows to use parallel MPI-I/O, thus the mesh format is designed for a fast parallel read-in, using large arrays. There is also the GUI *HDFView* to browse h5 files.

An important feature is that the elements are ordered along a space-filling curve. This allows a simple domain decomposition during parallel read-in, where one simply divides the number of elements by the number of domains, so that each domain is associated with a contiguous range of elements. That means one can directly start the parallel computation with an arbitrary number of domains ($\geq$ number of elements) and always read the same mesh file.

For each element, the neighbor connectivity information of the element sides and the element node information (index and position) are stored as a package per element, allowing to read contiguous data blocks for a given range of elements. To enable a fast parallel read-in, the coordinates of the same physical nodes are stored several times, but can be still associated by a unique global node index.

Notes:

- Indexing starts at 1! (Fortran Style)
- Element connectivity is based on CGNS unstructured mesh standard (CFD general notation system, http://cgns.sourceforge.net), see Section **??**.
- The polynomial degree N_g of the curved element mappings is globally defined. Straight-edged elements are found for $N_g = 1$.
- Only the nodes for the volume element mapping and no surface mappings are stored.
- Curved node positions in reference space are uniform for all element types (see Section **??**).
- Data types: we use 32bit INTEGER and 64bit REAL (double precision), if not stated differently.

2 File Description

HOPR generates **_mesh.h5* files. You can find examples of the mesh file by executing the tutorials in HOPR, and you can browse the files using *HDFView*.

2.1 Global Attributes

These attributes are defined globally for the whole mesh. For a mesh with elements having only straight edges, the polynomial degree of the element mapping is *Ngeo*= $N_g = 1$. A mesh with curved elements has a fixed polynomial degree $N_g > 1$ for all elements.

Attribute	Data type	Description
`Version`	REAL	Mesh File Version
`Ngeo` ≥ 1	INTEGER	Polynomial degree N_g of element mapping, used to determine the number of nodes per element
`nElems`	INTEGER	Total number of elements in mesh
`nSides`	INTEGER	Total number of sides (or element faces) in file
`nNodes`	INTEGER	Total number of nodes in file
`nUniqueSides`	INTEGER	Total number of geometrically unique sides in the mesh
`nUniqueNodes`	INTEGER	Total number of geometrically unique nodes in the mesh
`nBCs`	INTEGER	Size of the Boundary Condition list

Table 2.1: Mesh File attributes.

2.2 Data Arrays

The mesh information is organized in arrays. The **`ElemInfo`** array is the first to read, since it contains the data range of each element in the **`SideInfo`** and **`NodeCoords`**/ **`GlobalNodeIDs`** arrays.

Array Name	Description	Type	Size
Main information:			
`ElemInfo`	Start & End positions of element data in SideInfo / NodeCoords	INTEGER	(1:6,1:**`nElems`***)
`SideInfo`	Side Data / Connectivity information	INTEGER	(1:5,1:**`nSides`***)
`NodeCoords`	Node Coordinates	REAL	(1:3,1:**`nNodes`***)
`GlobalNodeIDs`	Globally unique node index	INTEGER	(1:**`nNodes`***)
`BCNames`	List of user-defined boundary condition names (max. 255 Characters)	STRING	(1:**`nBCs`**)
`BCType`	Four digit boundary condition code	INTEGER	(1:4,1:**`nBCs`**)
Additional information:			
`ElemBarycenters`	Barycenter location of each element	REAL	(1:3,1:**`nElems`***)
`ElemWeight`	Element Weights for domain decomposition (=1 by default)	REAL	(1:**`nElems`***)
`ElemCounter`	mesh statistics (no. of elements of each element type)	INTEGER	(1:2,1:11)

Table 2.2: List of all data arrays in mesh file. Dimensions marked with * will be distributed in parallel read mode.

2.3 Example Mesh

In the following sections, we explain the array definitions and show an example, which refers to the mesh in Fig. ?? with straight-edges, so $N_g = 1$. There is one element of each type, a tetrahedron, a pyramid, a prism and a hexahedron, four elements in total. Corner nodes and element sides have unique indices.

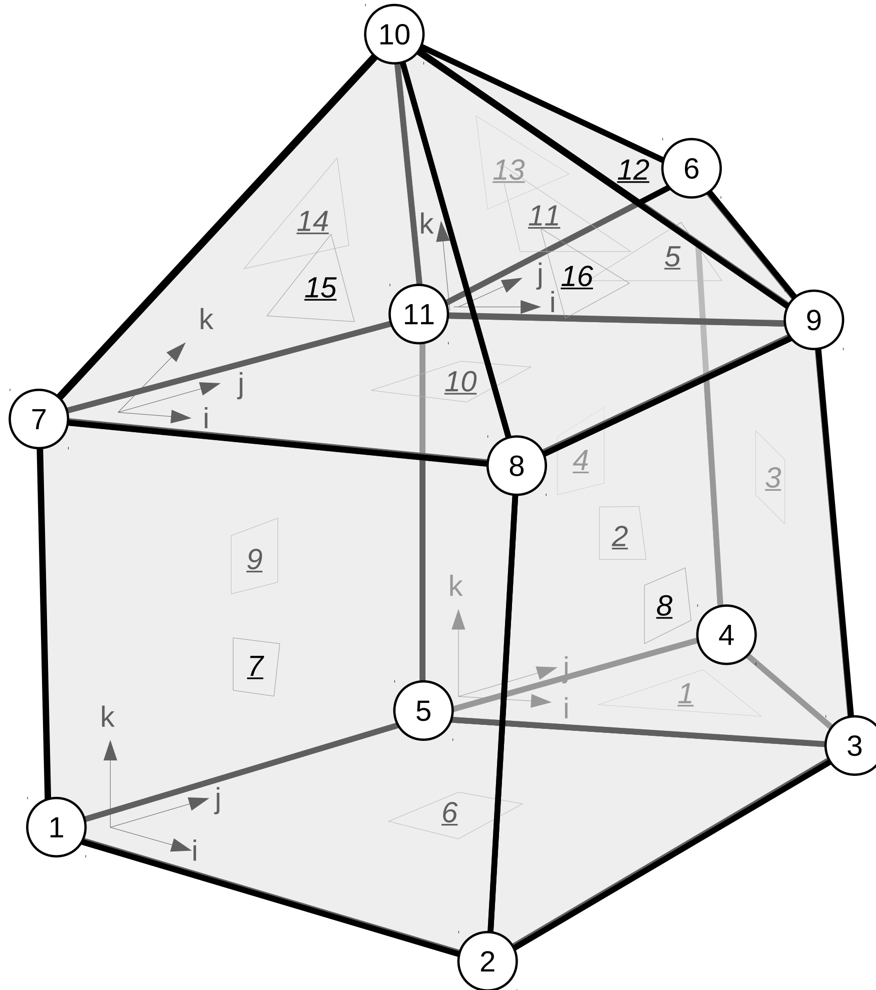


Figure 2.1: Example mesh with unique node IDs (circles) and unique side IDs (underline) and element-local coordinate system.

The global attributes of the mesh are

Ngeo	1	**nElems**	4 (Prism,Hex,Tet,Pyra)
nSides	20 (=5+6+4+5)	**nNodes**	23 (=6+8+4+5)
nUniqueSides	16	**nUniqueNodes**	11
nBCs	4		

Table 2.3: Global attributes for example mesh.

2.4 Array Definitions

2.4.1 Element Information (ElemInfo)

Name in file: **`ElemInfo`**
Type: INTEGER, Size: Array(1:6,1:**`nElems`***)
Description: Array containing elements, one element per row, **row number is elemID.**

The data is always stored elementwise, which results in storing it multiple times. However, this way, each processor has a defined, non overlapping, range of side /node information, where it can perform IO operations, minimizing the need of communication between processors.

	Element Type	*Zone*	*offsetIndSIDE*	*lastIndSIDE*	*offsetIndNODE*	*lastIndNODE*
1	116	1	0	5	0	6
2	118	1	5	11	6	14
3	104	2	11	15	14	18
4	115	2	15	20	18	23

Element Type:	Encoding for element type, seeSection **??**.
Zone:	Element group number.
offsetIndSIDE/lastIndSIDE:	Each element has a range of sides in the **`SideInfo`** array.
offsetIndNODE/lastIndNODE:	Each element has a range of node coordinates in the **`NodeCoords`** array and **`GlobalNodeIDs`** array for unique indices.

Table 2.4: `ElemInfo` array for example mesh Fig. **??** with 4 elements.

The range and the size are always defined as: ***Range=[offset+1,last], Size=last-offset***

The example shows the four different elements (prism/hexahedron/tetrahedra/pyramid), the prism and hexa are in zone 1 and the tet and the pyramid in zone 2. A detailed list of the element type encoding is found in Tab. **??**.

2.4.2 Side Information (SideInfo)

Name in file: **SideInfo**
Type: INTEGER, Size: Array(1:6,1:**nSides***)
Description: Side array, all information of one element is a set of all element sides (CGNS ordering, Fig. **??**).
offsetIndSIDE/lastIndSIDE in **ElemInfo** refers to the row index of one set of element sides.

	SideType	GlobalSideID	nbElemID	10*nbLocSide +Flip	BCID	in **ElemInfo**
1	3	1	0	0	1	(offsetIndSIDE+1,1)
2	14	2	2	43	0	
3	14	3	0	0	3	
4	14	4	0	0	4	
5	3	5	3	12	0	(lastIndSIDE,1)
6	14	6	0	0	1	(offsetIndSIDE+1,2)
7	14	7	0	0	2	
8	14	8	2	50	3	
9	14	-2	1	23	0	
10	14	9	2	30	4	
11	14	10	4	14	0	(lastIndSIDE,2)
12	3	-5	1	52	0	(offsetIndSIDE+1,3)
13	3	11	4	42	0	
14	3	12	0	0	3	
15	3	13	0	0	4	(lastIndSIDE,3)
16	14	-10	2	61	0	(offsetIndSIDE+1,4)
17	3	15	0	0	2	
18	3	16	0	0	3	
19	3	-11	3	22	0	
20	3	14	0	0	4	(lastIndSIDE,4)

SideType:	Side type encoding, the number of corner nodes is the last digit (triangle/quadrangle), more details see Section **??**.
GlobalSideID:	unique global side identifier, can be directly used as MPItag, negative if side is a slave side (a master and a slave side is defined for side connections).
nbElemID:	ElemID of neighbor element (= 0 for no connection). This helps to quickly build up element connections, for local (inside local element range) as well as inter-processor element connections.
*10*nbLocSide+Flip*:	first digit : local side of the connected neighbor element $\in [1, \dots, 6]$, last digit: Orientation between the sides (flip $\in [0, \dots, 4]$), seeSection **??**.
BCID:	Refers to the row index of the Boundary Condition List in **BCNames** / **BCType** array ($\in [1, \dots$ `nBCs`$]$). = 0 for inner sides. Note that $\neq 0$ for periodic and inner boundary conditions, while nbElemID and nbLocSide+Flip are given, see sec. **??**.

Table 2.5: SideInfo array for example mesh Fig. **??**.

2.4.3 Node Coordinates and Global Index

Name in file:	**NodeCoords**
Type:	REAL Size: Array(1:3,1:**nNodes***)
Description:	The coordinates of the nodes of the element, as a set for each element. *offsetIndNODE/lastIndNODE* in **ElemInfo** refers to the row index of one set of element nodes.

Name in file:	**GlobalNodeIDs**
Type:	INTEGER Size: Array(1:**nNodes***)
Description:	The unique global node identifier corresponding to the node at the same array position in **NodeCoords**.

The node list contains the high order nodes of the element, so the number of nodes per element depends on the polynomial degree of the element mapping N_g. From this list, the corner nodes can be extracted. The details of the node ordering are explained in Section **??**. It is important to note that in the case of $N_g = 1$, our node ordering does NOT correspond to the CGNS corner node ordering for pyramids and hexahedra. Note that the nodes are multiply stored because of the parallel I/O, and therefore the GlobalNodeID is needed for a unique node indexing.

NodeCoords		**GlobalNodeIDs**	in **ElemInfo**
$(x, y, z)_5$		5	(offsetIndNODE+1,1)
$(x, y, z)_3$		3	
$(x, y, z)_4$		4	
$(x, y, z)_{11}$		11	
$(x, y, z)_9$		9	
$(x, y, z)_6$		6	(lastIndNODE,1)
$(x, y, z)_1$		1	(offsetIndNODE+1,2)
$(x, y, z)_2$		2	
$(x, y, z)_5$		5	
$(x, y, z)_3$		3	
$(x, y, z)_7$		7	
$(x, y, z)_8$		8	
$(x, y, z)_{11}$		11	
$(x, y, z)_9$		9	(lastIndNODE,2)
$(x, y, z)_{11}$		11	(offsetIndNODE+1,3)
$(x, y, z)_9$		9	
$(x, y, z)_6$		6	
$(x, y, z)_{10}$		10	(lastIndNODE,3)
$(x, y, z)_7$		7	(offsetIndNODE+1,4)
$(x, y, z)_8$		8	
$(x, y, z)_{11}$		11	
$(x, y, z)_9$		9	
$(x, y, z)_{10}$		10	(lastIndNODE,4)

Table 2.6: **NodeCoords** and **GlobalNodeIDs** array for the example mesh Fig. **??**. The node ordering is explained in Section **??**.

2.4.4 Boundary Conditions

Name in file: **`BCNames`**
Type: STRING, Size: Array(1:**`nBCs`**)
Description: User-defined list of boundary condition names.

Name in file: **`BCType`**
Type: INTEGER, Size: Array(1:4,1:**`nBCs`**)
Description: User-defined array of 4 integers per boundary condition.

The boundary conditions are completely defined by the user. Each BCID from the **`SideInfo`** array refers to the **position** of the boundary condition in the **`BCNames`** list. An additional 4 integer code in **`BCType`** is available for user-defined attributes.

Ind	Boundary Conditions Name:		**`BCType`**
1	lowerWall		(4,0,0,0)
2	Inflow		(2,0,0,0)
3	OutflowRight		(10,0,0,0)
4	OutflowLeft		(8,0,0,0)

Table 2.7: **`BCNames`** and **`BCType`** array for the example mesh, representing a list of boundary condition names.

The **`BCType`** array consists of the following entries, of which some are specific to HOPR:

`BCType` = (*BoundaryType, CurveIndex, StateIndex, PeriodicIndex*)

BoundaryType:	Actual type of boundary condition (e.g. inflow, outflow, periodic). **Reserved values:** BoundaryType=1 is reserved for periodic boundaries and BoundaryType=100 is reserved for "inner" boundaries or "analyze sides". For these two cases the sides in the SideInfo array will have a neighbor side/element/flip specified, all other sides with BCs are not connected!
CurveIndex:	Geometry tag used to distinguish between multiple BCs of the same type, e.g. to specify the original CAD surface belonging to the mesh side. Also used to control some mesh curving features, sides with CurveIndex>0 are curved, while sides with CurveIndex=0 are mostly (bi-) linear.
StateIndex:	Specifies the index of a reference state to be used inside the solver. This value is completely used-defined and will not be used/checked/modified by HOPR.
PeriodicIndex:	Only relevant for periodic sides, ignored for others. For periodic connections two boundary conditions are required, having the same absolute PeriodicIndex, one with positive, the other with negative sign.

3 Parallel Read-in

The overall parallel read-in process is depicted in Fig. **??**. The Algorithms **??**,**??**,**??** describe how to open and close a HDF5 file and read the file attributes.

Each parallel process (MPI rank) has to read a contiguous element range, which will be basically defined by dividing the total number of elements by the number of domains, already leading to the domain decomposition. The element distribution is computed locally on each rank. Follow Algorithm **??** for an equal distribution of an arbitrary number of elements on an arbitrary number of domains/ranks. The algorithm is easy to extend to account for different element weights. The element distribution is saved in the *offsetElem* array of size *0:nDomains*. The element range for each domain (*mydom∈[0:nDomains-1]*) is then

ElementRange(myDom)=[offsetElem(myDom)+1;offsetElem(myDom+1)]

Note that the *offsetElem* array will have the information of all element ranges of all ranks, which is very helpful for building the inter-domain mesh connectivity to quickly find neighbor elements on other domains/ranks.

Using the number of local elements and the offset, we read the non-overlapping sub-arrays of the **`ElemInfo`** array in parallel (using hyperslab HDF5 commands, see Algorithm **??**), which will assign a continuous sub-array of element informations for each rank. With the local element informations, we easily compute the offset and size of sub-arrays for the side data (**`SideInfo`**) and node data (**`NodeCoords`**), by computing

firstElem = *offsetElem(myDom)+1*
lastElem = *offsetElem(myDom+1)*

firstSide = **`ElemInfo`***(offsetIndSIDE,firstElem)+1*
lastSide = **`ElemInfo`***(lastIndSIDE,lastElem)*

firstNode = **`ElemInfo`***(offsetIndNode,firstElem)+1*
lastNode = **`ElemInfo`***(lastIndNODE,lastElem)*

and again read the non-overlapping sub-arrays in parallel. Now element geometry is easily built locally. Local element connectivities would only have neighbor element indices inside the local element range and can directly be assigned. The overall read-in process is summarized in Algorithm **??**.

For the inter-domain connectivity, we have to find the domain containing the neighbor element. A quick search is done with a bisection of the *offsetElem* array, since element ranges are monotonically increasing, see Algorithm **??**.

Finally, we group the sides connected to each neighbor domain and sort the sides along the global side index (known from **`SideInfo`**). This creates the same side list on both domains without any communication. If an orientation of the side link is needed, the side is always marked either master or slave (positive or negative global side index).

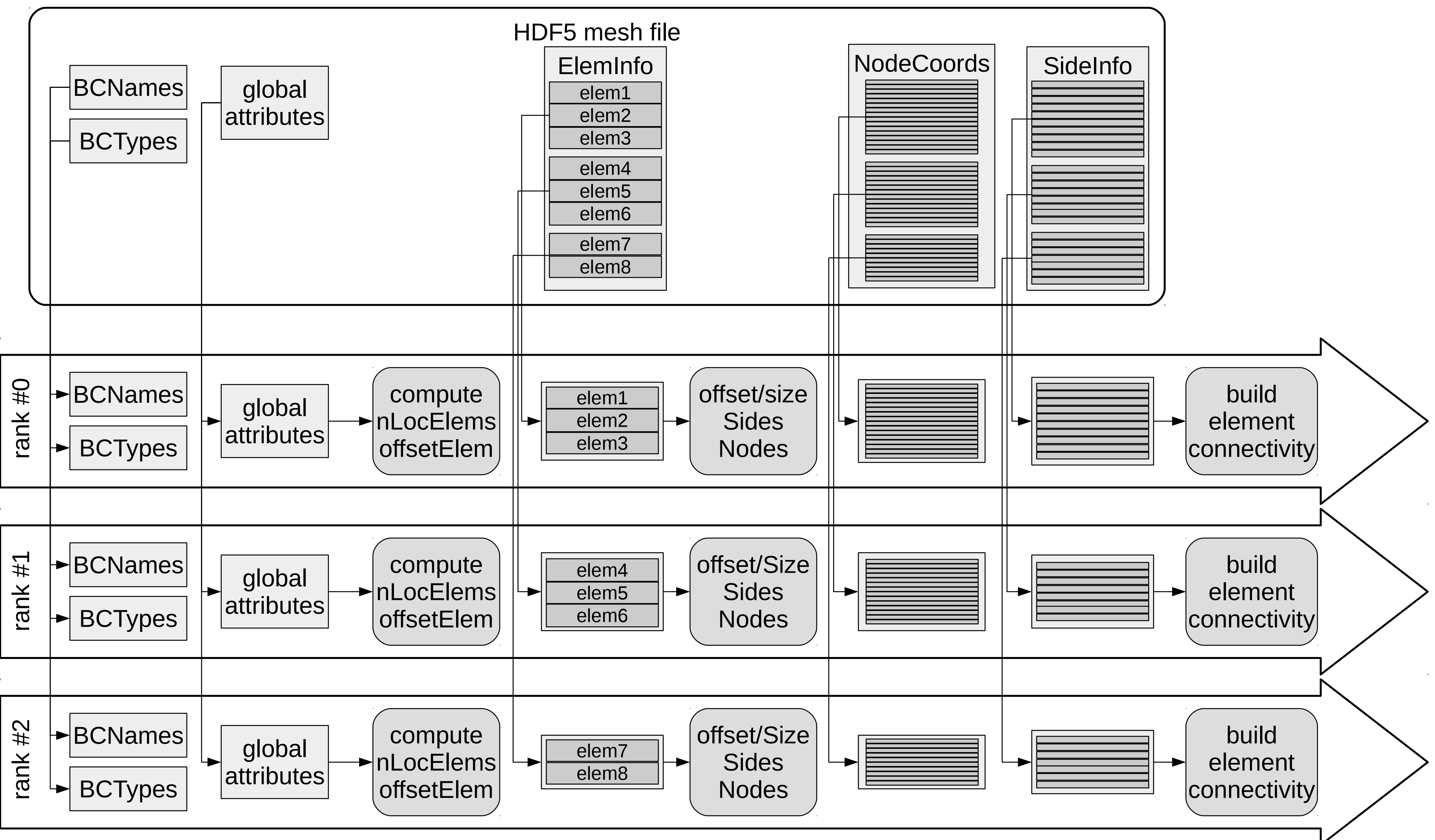


Figure 3.1: Parallel read-in process of the HDF5 mesh file, exemplary with 8 elements on 3 MPI ranks /domains.

Algorithm 1: Open HDF5 File for read access in parallel and setup file access property list with parallel I/O (MPI)

Procedure *OpenDataFile*

Input: FileName
Output: FileID

```
plist= H5Pcreate(H5P_FILE_ACCESS)
H5Pset_fapl_mpio(plist, MPI_COMM_WORLD, MPIInfo)
FileID = H5Fopen(FileName,H5F_ACC_RDWR,plist )
H5Pclose(plist)
```

Algorithm 2: Close HDF5 File

Procedure *CloseDataFile*

Input: FileID
Output:

```
H5Close(FileID)
```

Algorithm 3: Read attribute from File

Procedure *ReadAttribute*

Input: FileID, AttributeName, Dimsf(1)
Output: attribute

```
AttrID = H5Aopen(FileID, AttributeName)
typeID= H5Aget_type(DsetID)
H5Dread( AttrID, typeID, attribute )

H5Tclose(typeID)
H5Aclose(AttrID)
```

Algorithm 4: Simple Domain Decomposition: Assign local number of elements of domain *myDom* $\in [0; nDomains - 1]$ and element ranges for all domains

```
Procedure DomainDecomp
    Input: nElems, nDomains, myDom
    Output: offsetElem(0:nDomains)

    nLocalElems ← nElems/nDomains
    remainElems ← nElems-nLocalElems * nDomains
    for iDom = 0 to nDomains-1 do
        offsetElem(iDom)← iDom * nLocalElems + MIN(iDom,remainElems)
    offsetElem(nDomains)←nElems
```

Algorithm 5: Parallel non-overlapping read-in of an array. Note that arrays start a 0 in HDF5!

```
Procedure ReadArray
    Input: FileID, ArrayName, Rank, Dimsf(rank), offset(rank)
    Output: subarray

    MemSpace = H5Screate_simple(rank, Dimsf)
    DsetID = H5Dopen(FileID, ArrayName)
    FileSpace = H5Dget_space(DsetID)
    H5Sselect_hyperslab(FileSpace, H5Sselect_hyperslab, offset,Dimsf)
    plist= H5Pcreate(H5P_DATASET_XFER)                 /* create property list */
    H5Pset_dxpl_mpio(plist, H5FD_MPIO_COLLECTIVE)          /* collective read */
    typeID= H5Dget_type(DsetID)
    /* read local data array                                                 */
    H5Dread( DsetID, typeID, MemSpace, FileSpace, plist, subarray )

    H5Tclose(typeID)
    H5Pclose(plist)
    H5Sclose(FileSpace)
    H5Dclose(DSetID)
    H5Sclose(MemSpace)
```

Algorithm 6: Overall parallel read-in process for an MPI rank

```
Procedure ReadMesh
    Input: MeshFile,nRanks,myRank
    Output: ElemInfo,SideInfo,NodeCoords

    FileID= OpenDataFile(MeshFile)

    nGlobalElems= ReadAttribute(FileID,'nElems',1)
    offsetElem(0:nDomains)= DomainDecomp(nGlobalElems,nRanks,myRank)
    /* read local subarray of ElemInfo */
    firstElem= offsetElem(myRank)+1
    nLocalElems= offsetElem(myRank+1)-offsetElem(myRank)
    ElemInfo(1:6,1:nLocalElems)=
    ReadArray(FileID,'ElemInfo',2,(6,nLocalElems),(0,firstElem-1) )

    /* read local subarray of NodeCoords and GlobalNodeIDs */
    firstNode= ElemInfo(5,1)+1
    nLocalNodes = ElemInfo(6,nLocalElems)-ElemInfo(5,1)
    NodeCoords(1:3,1:nLocalNodes)=
    ReadArray(FileID,'NodeCoords',2,(3,nLocalNodes),(0,firstNode-1) )
    GlobalNodeIDs(1:nLocalNodes)=
    ReadArray(FileID,'GlobalNodeIDs',1,(nLocalNodes),(firstNode-1) )

    /* read local subarray of SideInfo */
    firstSide= ElemInfo(3,1)+1
    nLocalSides = ElemInfo(4,nLocalElems)-ElemInfo(3,1)
    SideInfo(1:5,1:nLocalSides)= ReadArray(FileID,'SideInfo',2,(5,nLocalSides),(0,firstSide-1) )

    CloseDataFile(FileID)
```

Algorithm 7: Find domain containing element index: use offsetElem array from Algorithm **??** and perform a bisection

```
Procedure ElemToRank
    Input: nDomains, offsetElem(0:nDomains), elemID
    Output: domain

    domain=0
    maxSteps ← INT(LOG(REAL(nDomains))/LOG(2))+1
    low ← 0
    up ← nDomains-1
    if offsetElem(low) < elemID ≤ offsetElemMPI(low+1) then
        domain← low
    else if offsetElem(up) < elemID ≤ offsetElem(up+1) then
        domain← up
    else
        for i = 1 to maxSteps do
            mid=(up-low)/2+low                                  /* bisection */
            if offsetElem(mid) < elemID ≤ offsetElem(mid+1) then
                domain=mid                                      /* index found */
                return
            else if elemID > offsetElem(mid+1) then
                low=mid+1                                       /* seek in upper half */
            else
                up=mid
```

4 Element Definitions

4.1 Element Types

The classification of the element types is given in Tab. ??. The last digit is always the number of corner nodes. The classification is geometrically motivated. The element has a linear mapping if $N_g = 1$ and the corner nodes are an affine transformation of the reference element corner nodes, whereas bilinear stands for the general straight-edged element with $N_g = 1$, and non-linear for the high order case $N_g \geq 1$.

For mesh file read-in, only the number of element corner nodes is important to distinguish the 3D elements, since the polynomial degree N_g is globally defined.

ElementType	Index	ElementType	Index	ElementType	Index
Triangle, linear	3	Tetrahedron, linear	104	Prism, bilinear	116
Quad, linear	4	Pyramid, linear	105	Hexahedron, bilinear	118
		Prism, linear	106	Tetrahedron, non-linear	204
Quad, bilinear	14	Hexahedron, linear	108	Pyramid, non-linear	205
Triangle, non-linear	23			Prism, non-linear	206
Quad, non-linear	24	Pyramid, bilinear	115	Hexahedron, non-linear	208

Table 4.1: Element type encoding.

4.2 Element High Order Nodes

In the arrays **`NodeCoords`** and **`GlobalNodeIDs`** (Section ??), the element high order nodes are found as a node list, $1, \dots, \ell, \dots M_{\text{elem}}$. The number of nodes for each element is defined by the element type and the polynomial degree N_g of the mapping and is listed in Tab. ??. See Section ?? if one needs only the corner nodes of the linear mesh.

Element Type:	#Corner nodes	#HO nodes (M_{elem})
Triangle	3	$\frac{1}{2}(N_g+1)(N_g+2)$
Quad	4	$(N_g+1)^2$
Tetrahedron	4	$\frac{1}{6}(N_g+1)(N_g+2)(N_g+3)$
Pyramid	5	$\frac{1}{6}(N_g+1)(N_g+2)(2N_g+3)$
Prism	6	$\frac{1}{2}(N_g+1)^2(N_g+2)$
Hexhedron	8	$(N_g+1)^3$

Table 4.2: Element node count.

The mapping from the node list to the node position

$$\ell \mapsto (i,j,k) \quad \ell \in [1; M_{\text{elem}}] \quad 0 \le i,j,k \le N_g \tag{4.1}$$

is defined by Algorithm **??** and an example is shown for quadratic mapping in Fig. **??**. The high order node positions are regular in reference space $-1 \le (\xi,\eta,\zeta) \le 1$ and therefore can be easily computed from the (i,j,k) index of the node ℓ by

$$(\xi,\eta,\zeta)_\ell = -1 + \frac{2}{N}(i,j,k)_\ell \tag{4.2}$$

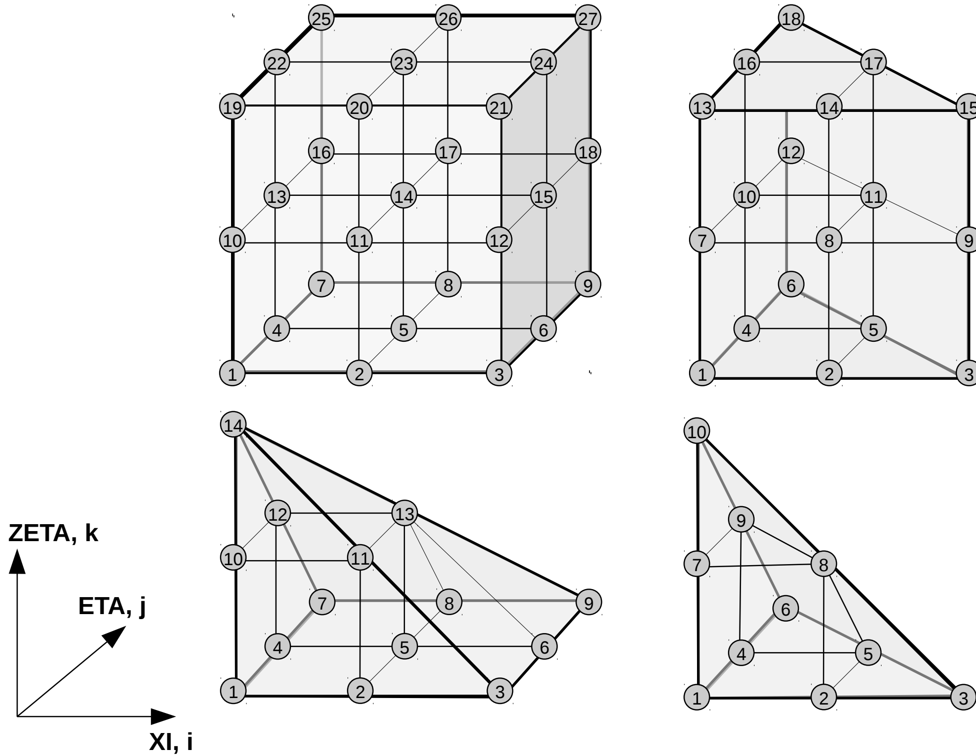


Figure 4.1: Example of the element high order node sorting from Algorithm **??** for a quadratic mapping ($N_g = 2$).

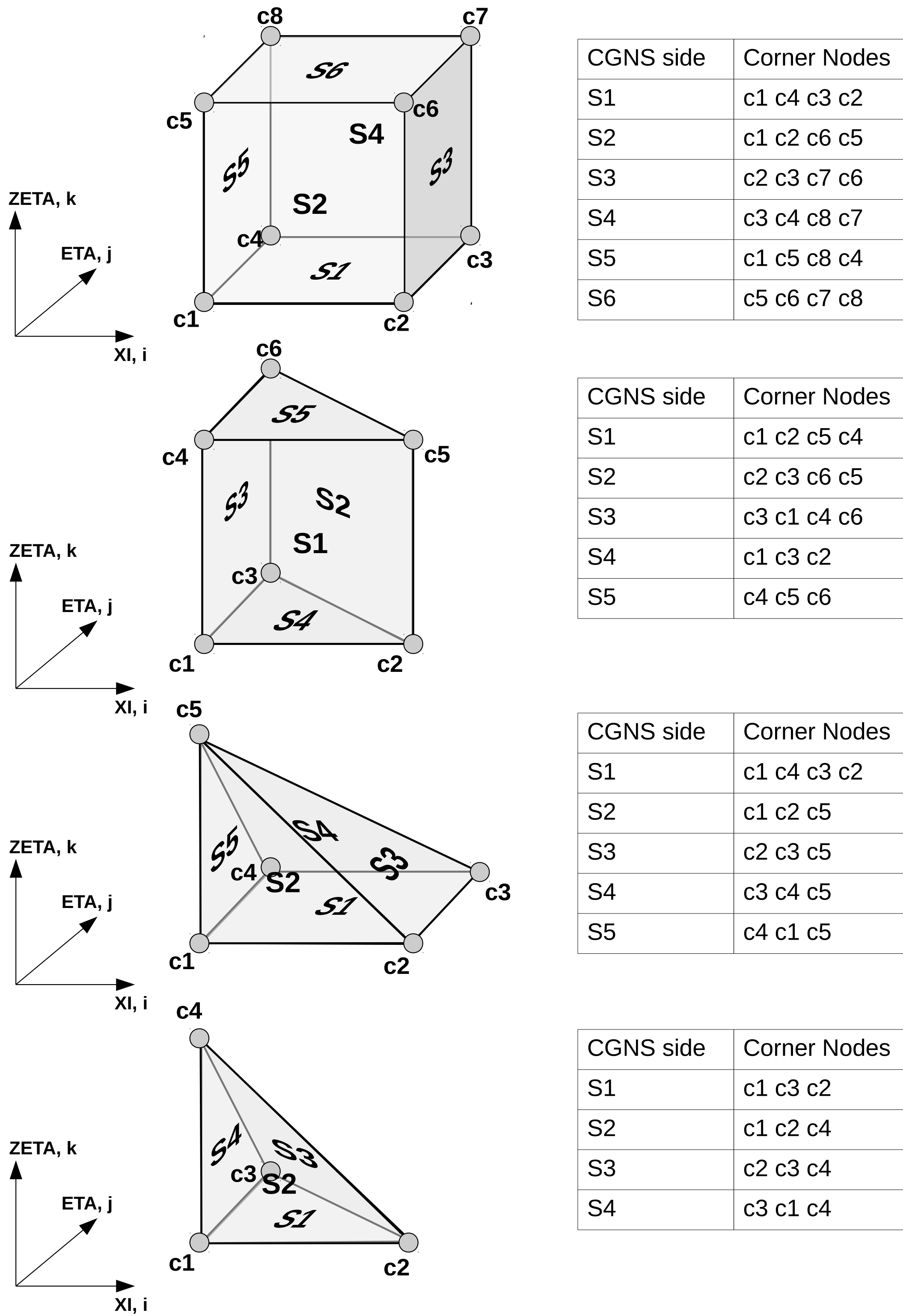
4.3 Element Corners, Sides

To define the element corner nodes, the side order and side connectivity, we follow the standard from CGNS SIDS (CFD General Notation System, Standard Interface Data Structures, http://cgns.sourceforge.net/). The definition is sketched in Fig. **??**. To get the CGNS corner nodes from the high order node list, follow Algorithm **??**. Note that in the case of $N_g = 1$, the node ordering does **not** correspond to the CGNS corner node ordering for pyramids and hexahedra!

Especially, the CGNS standard defines a local coordinate system of each element side. The side's first node will be the origin, and the remaining nodes are ordered in the direction of the outward pointing normal.

Algorithm 8: Mapping between the list of high order nodes to i,j,k positions: given the polynomial degree N and the number of corner nodes of the element

```
Procedure CurvedNodeMapping
    Input: N,nCornerNodes
    Output: nCurvedNodes,Map(3,nElemNodes),MapInv(0:N,0:N,0:N),refPos(3,nElemNodes)
    ℓ = 0
    switch nCornerNodes do
        case 4 (Tetrahedron)
            nCurvedNodes= (N + 1) * (N + 2) * (N + 3)/6
            for k = 0 to N do
                for j = 0 to N − k do
                    for i = 0 to N − j − k do
                        ℓ ← ℓ + 1
                        Map(:, ℓ) ← (i, j, k)
                        MapInv(i, j, k) ← ℓ
        case 5 (Pyramid)
            nCurvedNodes= (N + 1) * (N + 2) * (2N + 3)/6
            for k = 0 to N do
                for j = 0 to N − k do
                    for i = 0 to N − k do
                        ℓ ← ℓ + 1
                        Map(:, ℓ) ← (i, j, k)
                        MapInv(i, j, k) ← ℓ
        case 6 (Prism)
            nCurvedNodes= (N + 1) * (N + 1) * (N + 2)/2
            for k = 0 to N do
                for j = 0 to N do
                    for i = 0 to N − j do
                        ℓ ← ℓ + 1
                        Map(:, ℓ) ← (i, j, k)
                        MapInv(i, j, k) ← ℓ
        case 8 (Hexa)
            nCurvedNodes= (N + 1) * (N + 1) * (N + 1)
            for k = 0 to N do
                for j = 0 to N do
                    for i = 0 to N do
                        ℓ ← ℓ + 1
                        Map(:, ℓ) ← (i, j, k)
                        MapInv(i, j, k) ← ℓ
    for ℓ = 1 to nCurvedNodes do
        refPos(:, ℓ) ← -1+ 2/N * Map(:, ℓ)
```



CGNS side	Corner Nodes
S1	c1 c4 c3 c2
S2	c1 c2 c6 c5
S3	c2 c3 c7 c6
S4	c3 c4 c8 c7
S5	c1 c5 c8 c4
S6	c5 c6 c7 c8

CGNS side	Corner Nodes
S1	c1 c2 c5 c4
S2	c2 c3 c6 c5
S3	c3 c1 c4 c6
S4	c1 c3 c2
S5	c4 c5 c6

CGNS side	Corner Nodes
S1	c1 c4 c3 c2
S2	c1 c2 c5
S3	c2 c3 c5
S4	c3 c4 c5
S5	c4 c1 c5

CGNS side	Corner Nodes
S1	c1 c3 c2
S2	c1 c2 c4
S3	c2 c3 c4
S4	c3 c1 c4

Figure 4.2: Definition of corner nodes, side order and side orientation, from CGNS SIDS.

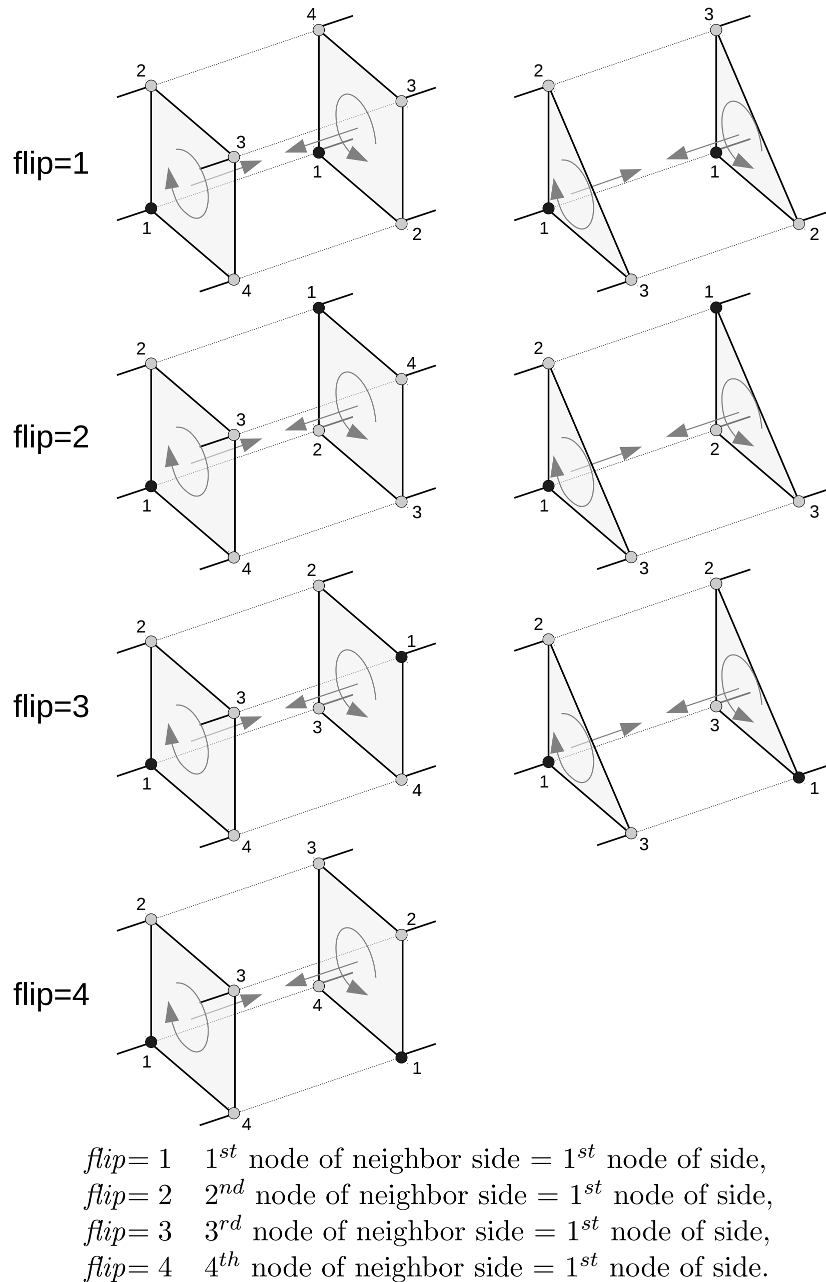
Algorithm 9: Mapping between the element curved node list and the CGNS corner nodes

```
Procedure CornerNodeMapping
    Input: N,nCornerNodes [,MapInv]
    Output: CGNSCornerMap(nCornerNodes)

    switch nCornerNodes do
        case 4 (Tetrahedron)
            CGNSCornerMap(1)  ← 1                          [=MapInv(0,0,0)]
            CGNSCornerMap(2)  ← (N+1)                      [=MapInv(N,0,0)]
            CGNSCornerMap(3)  ← (N+1)*(N+2)/2              [=MapInv(0,N,0)]
            CGNSCornerMap(4)  ← (N+1)*(N+2)*(N+3)/6        [=MapInv(0,0,N)]
        case 5 (Pyramid)
            CGNSCornerMap(1)  ← 1                          [=MapInv(0,0,0)]
            CGNSCornerMap(2)  ← (N+1)                      [=MapInv(N,0,0)]
            CGNSCornerMap(3)  ← (N+1)**2                   [=MapInv(N,N,0)]
            CGNSCornerMap(4)  ← N*(N+1)+1                  [=MapInv(0,N,0)]
            CGNSCornerMap(5)  ← (N+1)*(N+2)*(2*N+3)/6      [=MapInv(0,0,N)]
        case 6 (Prism)
            CGNSCornerMap(1)  ← 1                          [=MapInv(0,0,0)]
            CGNSCornerMap(2)  ← (N+1)                      [=MapInv(N,0,0)]
            CGNSCornerMap(3)  ← (N+1)*(N+2)/2              [=MapInv(0,N,0)]
            CGNSCornerMap(4)  ← N*(N+1)*(N+2)/2+ 1         [=MapInv(0,0,N)]
            CGNSCornerMap(5)  ← N*(N+1)*(N+2)/2+(N+1)      [=MapInv(N,0,N)]
            CGNSCornerMap(6)  ← (N+1)**2*(N+2)/2           [=MapInv(0,N,N)]
        case 8 (Hexa)
            CGNSCornerMap(1)  ← 1                          [=MapInv(0,0,0)]
            CGNSCornerMap(2)  ← (N+1)                      [=MapInv(N,0,0)]
            CGNSCornerMap(3)  ← (N+1)**2                   [=MapInv(N,N,0)]
            CGNSCornerMap(4)  ← N*(N+1)+1                  [=MapInv(0,N,0)]
            CGNSCornerMap(5)  ← N*(N+1)**2+1               [=MapInv(0,0,N)]
            CGNSCornerMap(6)  ← N*(N+1)**2+(N+1)           [=MapInv(N,0,N)]
            CGNSCornerMap(7)  ← (N+1)**3                   [=MapInv(N,N,N)]
            CGNSCornerMap(8)  ← N*(N+1)*(N+2)+1            [=MapInv(0,N,N)]
```

4.4 Element Connectivity

In the **`SideInfo`** array, we explicitly store the side-to-side connectivity information between elements, consisting of the neighbor element ID, the local side of the neighbor and the orientation, encoded with the variable *flip*. Using the local side system, the orientation between elements boils down to three cases for a triangular element side and four for a quadrilateral element side. The definition is given in Fig. **??**. Also note that the flip is symmetric, having the same value if seen from the neighbor side.



flip= 1 1st node of neighbor side = 1st node of side,
flip= 2 2nd node of neighbor side = 1st node of side,
flip= 3 3rd node of neighbor side = 1st node of side,
flip= 4 4th node of neighbor side = 1st node of side.

Figure 4.3: Definition of the orientation of side-to-side connection (*flip*) for quadrilateral and triangular element sides, the numbers are the local order of the element side nodes, as defined in Section **??**.

5 Additional Extensions: Hanging node interface

For complex geometries it is often desirable to use elements with hanging nodes to provide more geometric flexibility when meshing. As geometric restrictions are most severe for pure hexahedral meshes, the HOPR format supports a limited octree-like topology with hanging nodes for purely hexahedral meshes. The octree topology is implemented as extension to the existing mesh structure. While full octrees permit an element-side to have an arbitrary number of neighbors on various octree levels, our format supports only one octree level difference between element sides with two (anisotropic) and four neighbors, the single types are depicted in Figure **??**.

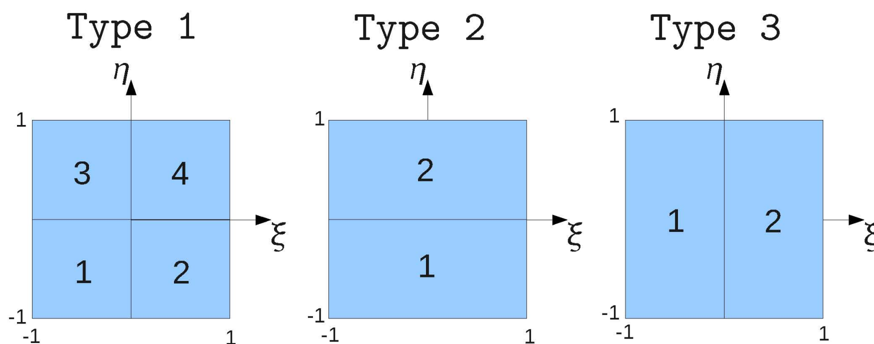


Figure 5.1: Possible types of mortar interfaces, with ξ, η denoting the sides local parameter space

For the connection of two elements over an octree level additional interfaces are required, which are termed mortar interfaces and are depicted in Figure **??**. Thereby the big side is denoted big mortar master side, the intermediate sides are denoted small mortar master sides. The sides of the small elements are denoted slave sides. Note that they do not require any information about the mortar interface and therefore the interface is only represented from the big element side in the data format.

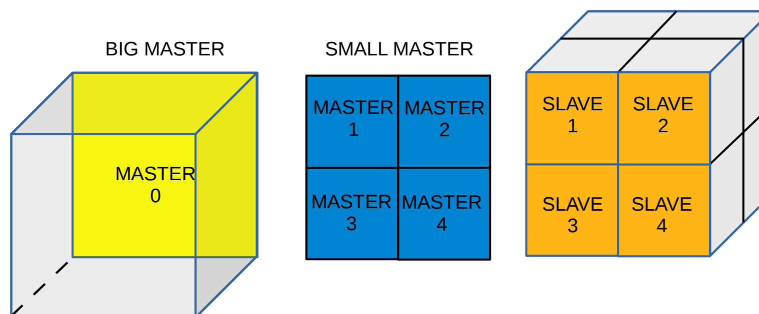


Figure 5.2: Structure of the mortar interface, with the local mortar ID defined from $0 - 4$

5.1 Changes to existing data format

The following differences are present for the ElemInfo array:

- **ElemInfo:** The range of sides defined by offsetIndSide and lastIndSide now includes the small mortar master sides.
- **SideInfo:** The field nbElemID of the big mortar master defines no connection to the neighbor element, but contains the type of the mortar interface from Figure **??** with negative sign. The type of the interface defines the number of the small mortar master sides.
- **SideInfo:** Only the small mortar masters have a valid nbElemID, defining the connection to the adjacent small elements.
- **SideInfo:** The list of sides belonging to an element includes the small mortar master sides sorted as exemplified in table **??**.
- **SideInfo:** (Mortar) Master sides always have flip=0, thus the small element sides are always slave sides.

Global SideID	local SideID	local MortarID
42	1	0
43	1	1
44	1	2
45	1	3
46	1	4
47	2	0
48	3	0
49	3	1
50	3	2
51	4	0
52	5	0
53	6	0

Table 5.1: Sorting example for sides in SideInfo, for an element containing two mortar interfaces of type 1 and type 2/3. Note that local SideID and MortarID are not stored in SideInfo.

5.2 Additional information for octrees

In addition to the existing format defined above, the mortar format contains non-necessary additional information concerning the octrees. It contains the octree node coordinates and a mapping of the element to the octrees. Note that the polynomial degree of the element mapping is defined as N_g, while the octrees may have an independent polynomial degree $N_{g,tree}$. Elements and trees can be identical in case the element is on the lowest octree level and the polynomial degrees are identical.

5.2.1 Global attributes

In addition to the global attributes defined in **??**, the non-conforming mesh format includes the following attributes.

Attribute	Data type	Description
IsMortarMesh	INTEGER	Identify mesh as a mortar mesh, if present
NgeoTree ≥ 1	INTEGER	Polynomial degree $N_{g,tree}$ of tree mapping, used to determine the number of nodes per element
nTrees	INTEGER	Total number of octrees
nNodesTree	INTEGER	Total number of tree nodes: $(N_{g,tree}+1)^3 \cdot nTrees$

Table 5.2: Additional mesh file attributes for octrees.

5.2.2 Mapping of the global element index (ElemID) to the octree index (TreeID)

Name in file: **ElemToTree**
Type: INTEGER Size: Array(1:**nElems***)
Description: The mapping from the global element index (ElemID) to its corresponding octree index (TreeID) if applicable.

5.2.3 Element bounds in tree reference space

Name in file: **xiMinMax**
Type: REAL Size: Array(1:3,1:2,1:**nElems***)
Description: The array contains the element bounds in the tree reference space $[-1,1]^3$ given by the minimum (1:3,1,ElemID) and maximum (1:3,2,ElemID) corner nodes.

5.2.4 Node Coordinates of the octrees

Name in file: **TreeCoords**
Type: REAL Size: Array(1:3,**nNodesTree***)
Description: The coordinates of the nodes of the tree, as a set for each tree.