

pumiMBBL-GPU with Kokkos

Library Documentation

Vignesh Vittal-Srinivasaragavan
RPI-SCOREC

October 20, 2021

1 New PUMI Data structure

1.1 Submesh-level – base classes and derived classes

```
class Submesh{
public:
    double xmin;
    double xmax;
    int Nel;
    double t0;
    double r;
    int Nel_cumulative;
    // enum type 0x01-uniform, 0x02-minBL and 0x04-maxBL
    Meshtype meshtype;
    Kokkos::View<double*> BL_coords; // BL coords on device
    double *host_BL_coords; // BL coords on host
    Submesh(...):...{}; //Constructor to submesh class
    // Virtual functions on host
    virtual int locate_cell_host(); // using analytical expressions
    virtual int update_cell()_host; // using adjacency search
    virtual void calc_weights_host(); // using stored node coordinates
    // Virtual functions not implemented on device
    // device copies of same routines implemented
    // with switch statements
};
```

The submesh class is identical to the `struct pumi_submesh` we have currently. Only difference is that the BL coordinates array is allocated with Kokkos type. We will have three derived classes for `pumi_submesh`, one for each type of meshing.

Uniform meshing:

```
class Uniform_Submesh : public Submesh{
public:
    Uniform_Submesh(...):...{}; // Constructor to uniform submesh cl
    // uniform mesh APIs on host
    int locate_cell_host(); // using analytical expressions
```

```

    int update_cell_host(); // using adjacency search
    void calc_weights_host();
    .
    .
    .
    // uniform mesh APIs on device
    KOKKOS_FUNCTION
    int locate_cell();
    KOKKOS_FUNCTION
    int update_cell();
    KOKKOS_FUNCTION
    void calc_weights();
    .
    .
    .
};

```

Left/Bottom BL meshing:

```

class MinBL_Submesh : public Submesh{
public:
    MinBL_Submesh(...):.....{}; // Constructor to minBL submesh class
    // minBL mesh APIs on host
    void locate_cell_host(); // using analytical expressions
    void update_cell_host(); // using adjacency search
    void calc_weights_host();
    .
    .
    .
    // minBL mesh APIs on device
    KOKKOS_FUNCTION
    int locate_cell();
    KOKKOS_FUNCTION
    int update_cell();
    KOKKOS_FUNCTION
    void calc_weights();
};

```

Right/Top BL meshing:

```

class MaxBL_Submesh : public pumi_submesh{
public:
    MaxBL_Submesh(...):.....{}; // Constructor to maxBL submesh class
    // maxBL mesh APIs on host
    void locate_cell_host(); // using analytical expressions
    void update_cell_host(); // using adjacency search
    void calc_weights_host();
    .
    .
    .
    // maxBL mesh APIs on device
    KOKKOS_FUNCTION

```

```

    int locate_cell();
    KOKKOS_FUNCTION
    int update_cell();
    KOKKOS_FUNCTION
    void calc_weights();
    .
    .
    .
};

```

This allows us to use the polymorphism feature to define different particle APIs with same name essentially replacing the function pointer data structures we previously implemented.

1.2 Mesh-level classes

```

class MeshOffsets{
// For 2D Mesh with inactive blocks
// Offsets keeps track of nodes/elements in inactive blocks
// Helps speed up computing global element/node ID
public:
    bool is_fullmesh; // Is mesh with no inactive blocks

    // aux data structure to compute nodeoffset - On device
    Kokkos::View<int**> nodeoffset_start;
    Kokkos::View<int**> nodeoffset_skip_bot;
    Kokkos::View<int**> nodeoffset_skip_mid;
    Kokkos::View<int**> nodeoffset_skip_top;
    // aux data structure to compute element offsets - On device
    Kokkos::View<int**> elemoffset_start;
    Kokkos::View<int*> elemoffset_skip;

    // Host copies of aux data structures
    int** host_nodeoffset_start;
    int** host_nodeoffset_skip_bot;
    int** host_nodeoffset_skip_mid;
    int** host_nodeoffset_skip_top;
    int** host_elemoffset_start;
    int* host_elemoffset_skip;

    int Nel_total; // Total active elements in domain
    int Nnp_total; // Total active nodes in domain

    MeshOffsets(){}; // Class constructor
};

class MeshBdry{
// For 2D Mesh with inactive blocks
// Stores block-level boundary edge/vertex information
public:
    // Block edge boundary info
    Kokkos::View<bool*> is_bdry_edge;

```

```

Kokkos::View<Vector3*> bdry_edge_normal;
// Block vertex boundary info
Kokkos::View<bool*> is_bdry_vert;
Kokkos::View<Vector3*> bdry_vert_normal;
// Starting boundary face ID on each boundary edge
Kokkos::View<int*> edge_to_face;

    // Host copies of same info
bool* host_is_bdry_edge;
Vector3* host_bdry_edge_normal;
bool* host_is_bdry_vert;
Vector3* host_bdry_vert_normal;
int *host_edge_to_face;

int Nbdry_faces; // number of boundary element faces

MeshBdry(){}; // Class constructor

};

class Mesh{
public:
    int ndim; // dimension
    int nsubmesh_x1; // number of x1-blocks
    int nsubmesh_x2; // number of x2-blocks
    Kokkos::View<bool**> isactive; // block activity info on device
    bool **host_isactive; // host copy of block activity
    int Nel_total_x1; // number of x1-elements
    int Nel_total_x2; // number of x2-elements
    int Nel_total; // Total mesh elements
    int Nnp_total; // Total mesh nodes
    MeshOffsets offsets; // object to MeshOffsets class
    MeshBdry bdry; // object to MeshBdry class
    Mesh(...):...{}; // Class constructor
};

```

1.3 Typedefs for submesh objects

Memory for submesh are dynamically allocated based on number of blocks. Hence copies are kept on both device and host.

```

// Submesh object array allocated in GPU using Kokkos View
// cannot be copied to CPU thru Kokkos::deep_copy()
using SubmeshDeviceViewPtr = Kokkos::View<DevicePointer<Submesh*>>;
// Copy of Submesh object array allocated in CPU
using SubmeshHostViewPtr = Submesh*;
// Convenience class for 3D-vectors of double
class Vector3;

```

1.4 Wrapper struct containing mesh and submesh objects

```

struct MBBL{
    Mesh mesh; // mesh obj
    SubmeshDeviceViewPtr submesh_x1; // x1-submesh obj in GPU
    SubmeshHostViewPtr host_submesh_x1; // x1-submesh obj in CPU
    SubmeshDeviceViewPtr submesh_x2; // x2-submesh obj in GPU
    SubmeshHostViewPtr host_submesh_x2; // x2-submesh obj in CPU
};

```

Object to this structure will be used in hpic2 and passed around in mesh APIs

2 PUMI Mesh initiation

Mesh initiation is still done through in-memory data structure `pumi_inputs`. In hPIC the mesh initiation is performed as

```

// Declare mesh inputs object
pumi::Mesh_Inputs *pumi_inputs;
// allocate memory
pumi_inputs = pumi::inputs_allocate(nsubmesh);
.
.
// parse inputs (from file or commandline) and populate pumi_inputs dat
.
.
// Set options for mesh
pumi::Mesh_Options pumi_options;
pumi_options.BL_storage_option = pumi::store_BL_coords_ON;
// Construct the final pumi wrapper struct
pumi::MBBL pumi_obj = pumi::initialize_MBBL_mesh(pumi_inputs, pumi_opti
// free pumi inputs memory
pumi::inputs_deallocate(pumi_inputs);

```

3 Element/Node ID conventions

Submesh IDs

A 2D submesh block is indexed by two integers (`isub`, `jsub`)

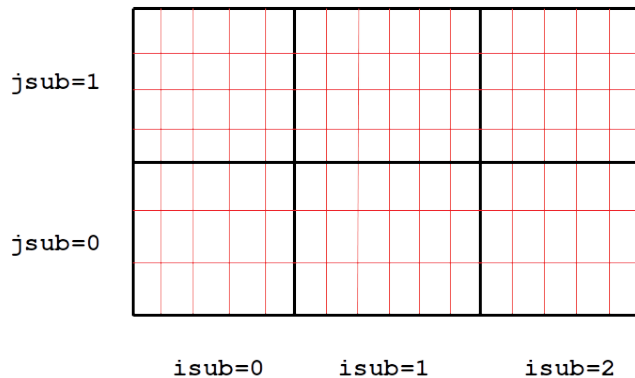


Figure 1: `isub` – submesh ID in x1-direction
`jsub` – submesh ID in x2-direction

Possible values:

$$0 \leq \text{isub} \leq N_{x_1}^{\text{submesh}} - 1 \quad 0 \leq \text{jsub} \leq N_{x_2}^{\text{submesh}} - 1$$

$N_{x_1}^{\text{submesh}}, N_{x_2}^{\text{submesh}}$ – Number of submesh blocks in x1, x2 directions respectively. Alternatively, the pairwise indices can also be flattened into one unique index and written as, $\text{flattened_sub_ID} = \text{jsub} \times N_{x_1}^{\text{submesh}} + \text{isub}$

Local Cell IDs

A cell inside a submesh block is indexed by two integers (icell , jcell)

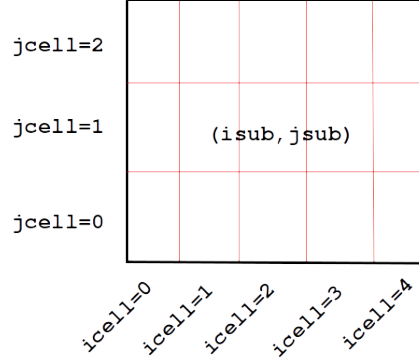


Figure 2: icell – local cell ID in x1-direction
 jcell – local cell ID in x2-direction

Possible values:

$$0 \leq \text{icell} \leq N_{x_1, \text{isub}}^{\text{el}} - 1 \quad 0 \leq \text{jcell} \leq N_{x_2, \text{jsub}}^{\text{el}} - 1$$

$N_{x_1, \text{isub}}^{\text{el}}$ – Number of elements along x1-direction in x1-block with ID isub

$N_{x_2, \text{jsub}}^{\text{el}}$ – Number of elements along x2-direction in x2-block with ID jsub

Alternatively, the pairwise indices can also be flattened into one unique index and written as,

$$\text{flattened_cell_ID} = \text{jcell} \times N_{x_1, \text{isub}}^{\text{el}} + \text{icell}$$

Component-wise Global Cell IDs

Same as global row/column index of a cell in rectilinear mesh

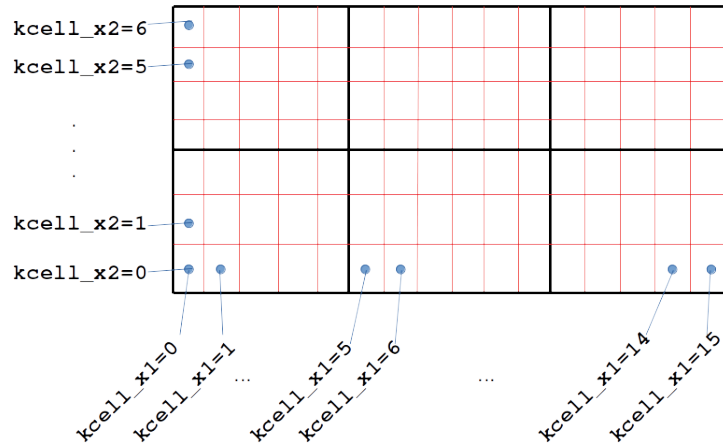


Figure 3: kcell_x1 – component-wise global cell ID in x1-direction
 kcell_x2 – component-wise global cell ID in x2-direction

Possible values:

$$0 \leq \text{kcell_x1} \leq N_{x_1}^{el,tot} - 1 \quad 0 \leq \text{kcell_x2} \leq N_{x_2}^{el,tot} - 1$$

$N_{x_1}^{el,tot}, N_{x_2}^{el,tot}$ – Number of total elements along x_1 , x_2 directions respectively

Component-wise Global Node IDs

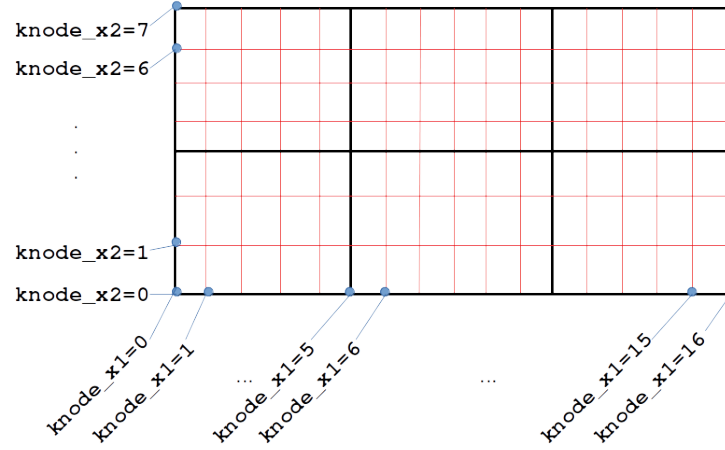


Figure 4: **knode_x1** – component-wise global cell ID in x_1 -direction
knode_x2 – component-wise global cell ID in x_2 -direction

Possible values:

$$0 \leq \text{knode_x1} \leq N_{x_1}^{el,tot} \quad 0 \leq \text{knode_x2} \leq N_{x_2}^{el,tot}$$

4 2D Global Element and Node conventions

4.1 Full Mesh (No Inactive Blocks)

2D Element Numbering: For a mesh with no inactive blocks the figure below show how the elements are numbered

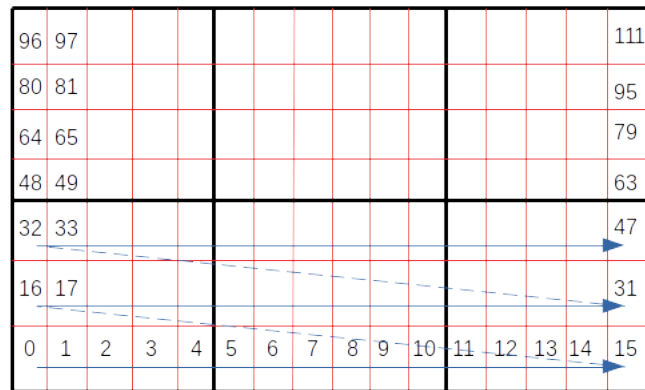


Figure 5: $N_{x_1}^{submesh} = 3, N_{x_1}^{el,tot} = 16$ $N_{x_2}^{submesh} = 2, N_{x_2}^{el,tot} = 7$

The global cell ID in 2D space `global_2D_cell` is computed from component-wise global cell IDs (`kcell_x1`, `kcell_x2`) i.e. global row and column IDs as

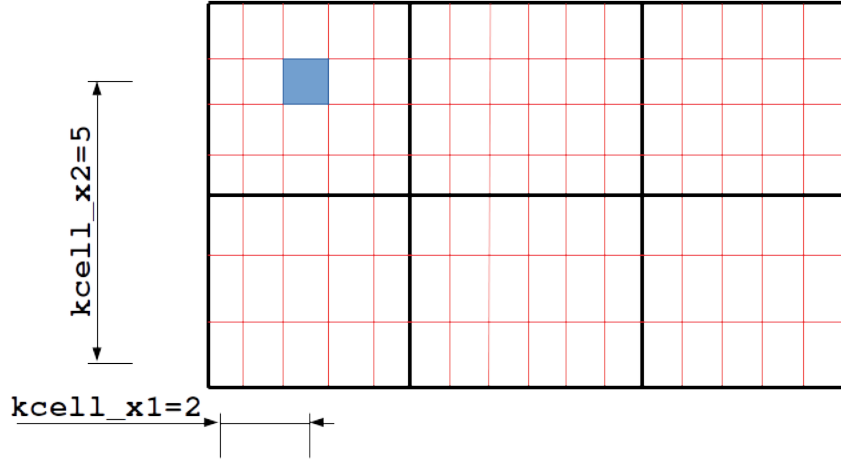


Figure 6: $\text{global_2D_cell} = \text{kcell_x2} \times N_{x_1}^{el,tot} + \text{kcell_x1} = 5(16) + 2 = 82$

Possible values:

$$0 \leq \text{global_2D_cell} \leq N_{2D}^{el,tot} - 1$$

$N_{2D}^{el,tot}$ – Total number of elements in the 2D domain.

On a full mesh $N_{2D}^{el,tot} = N_{x_1}^{el,tot} \times N_{x_2}^{el,tot}$

2D Node Numbering: For a mesh with no inactive blocks the figure below show how the nodes are numbered

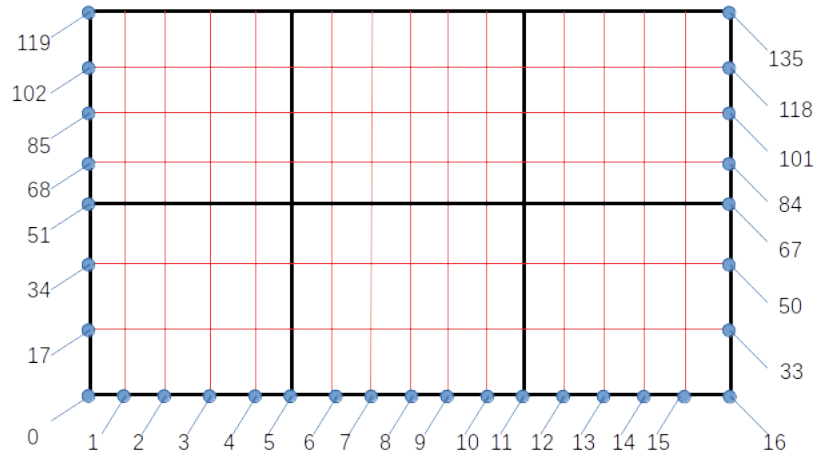


Figure 7: Node numbering is done similar to element numbering

elemoffset[isub][kcell_x2]															
18					18					18					
18					18					18					
18					18					18					
18					18					18					
12										18					
6										12					
0										6					

Element ID calculation – Example

The global element ID can be computed using the same expression used in full mesh and subtracting an offset (which tracks the number of skipped elements)

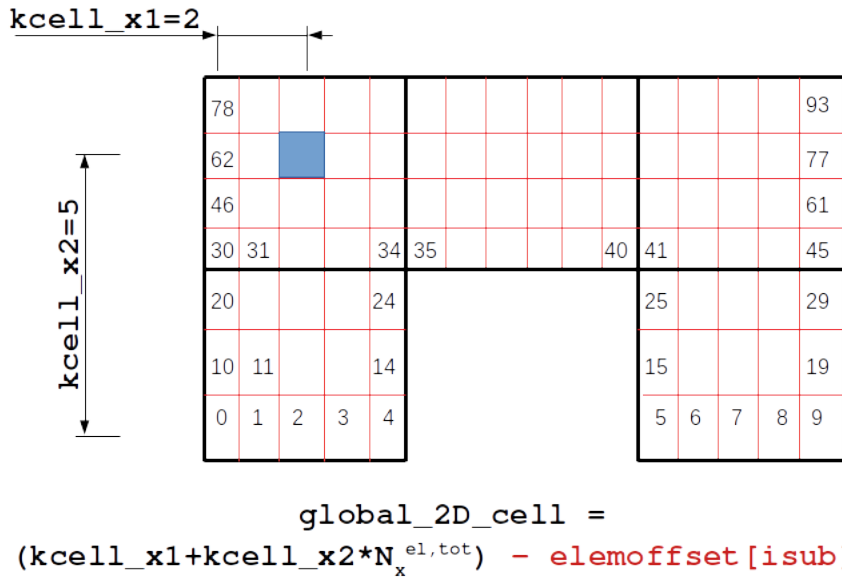
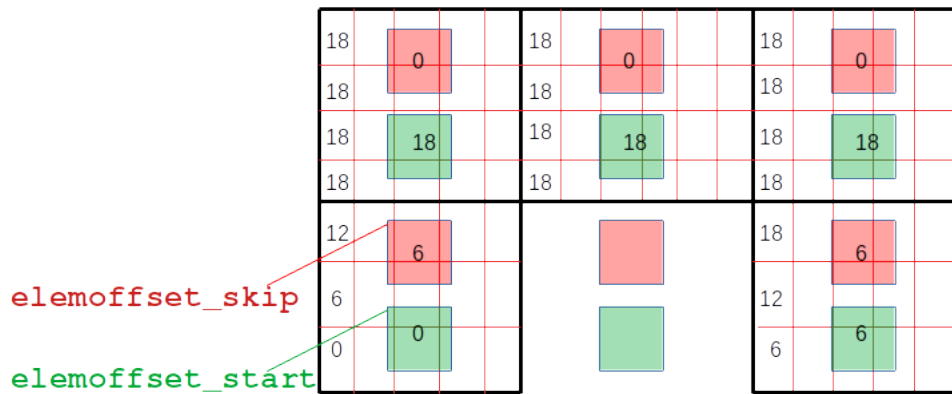


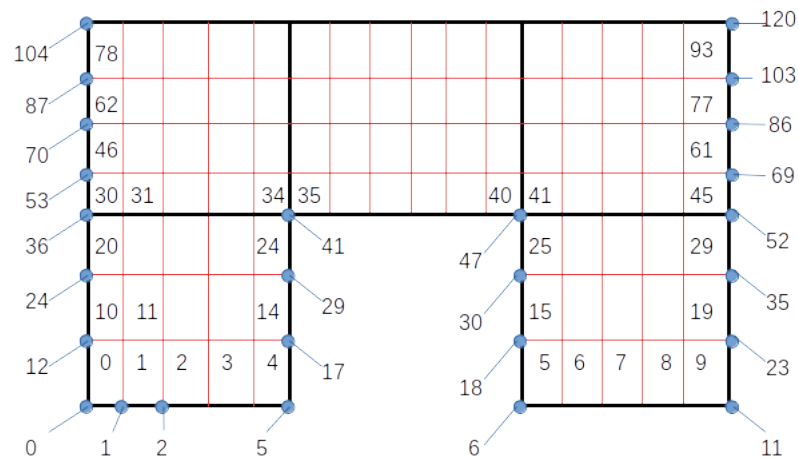
Figure 9: $global_2D_cell = 2 + 5 \times 16 - 18 = 64$

Computation of elemoffset without explicit storage The value of offset for each row in a x1-submesh block can be obtained by storing two additional integers for every submesh block

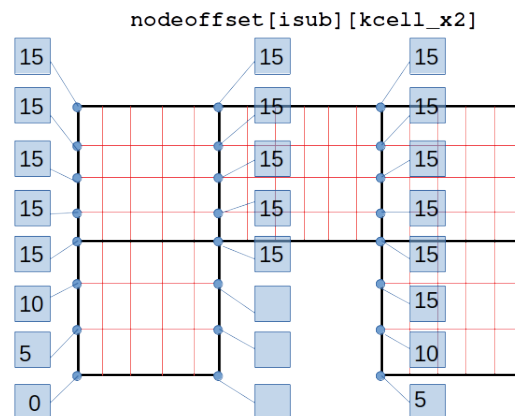
```
elemoffset[isub][kcell_x2] =
elemoffset_start[isub][jsub] + jcell*elemoffset_skip[isub][jsub]
```



2D Node Numbering: For a mesh with inactive blocks the figure below show how the nodes are numbered



Same as element numbering, we skip the nodes in the inactive blocks while numbering. However, the offset that keeps track of skipped nodes DO NOT follow the same 'nice' pattern inside a submesh block as the element offsets do.



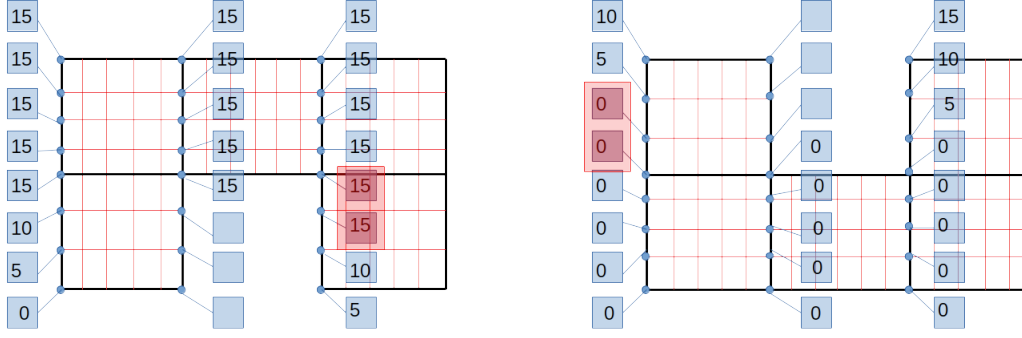
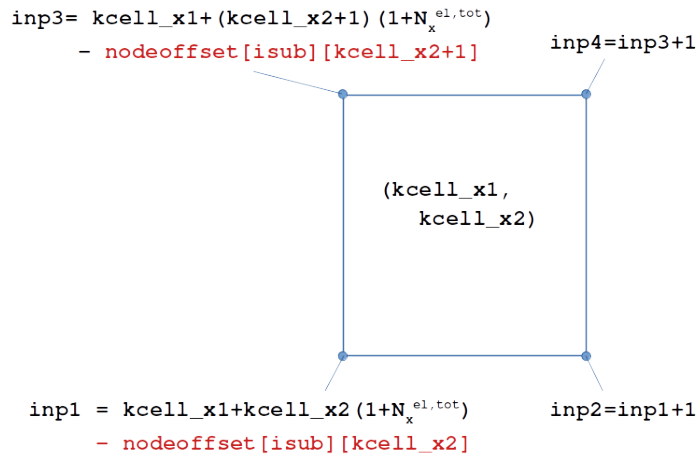


Figure 10: Examples where `nodeoffset` do not follow the same pattern as the rest of the block



Node ID calculation – Examples

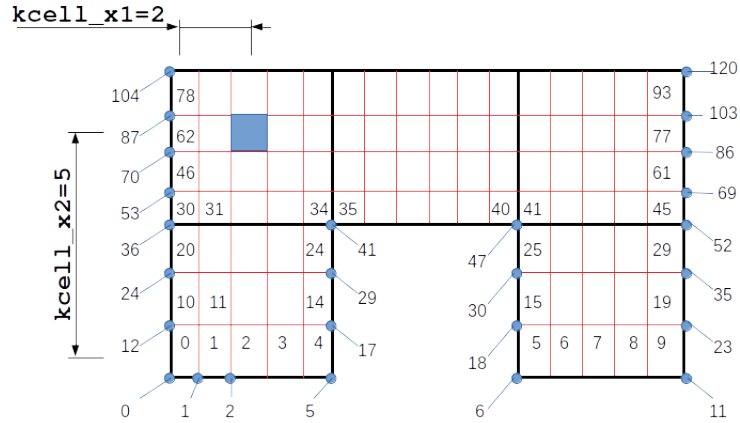


Figure 11: $\text{inp1}=2+5 \times 17-15=72$
 $\text{inp3}=2+6 \times 17-15=89$

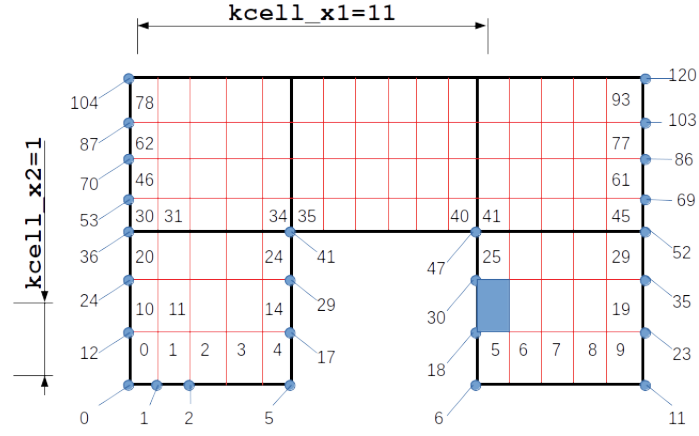
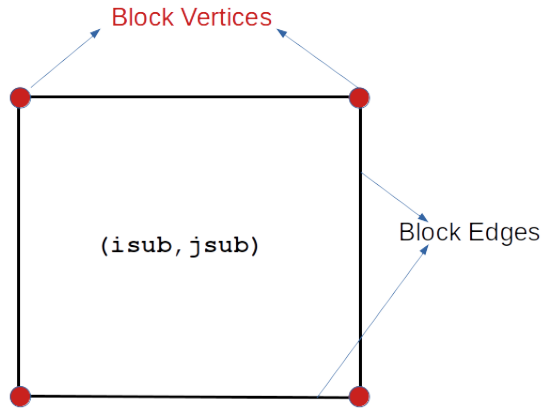


Figure 12: $\text{inp1}=11+1 \times 17-10=18$
 $\text{inp3}=11+2 \times 17-15=30$

NOTE: The auxiliary data structures `nodeoffset`, `elemoffset_start`, `elemoffset_skip` are all allocated as kokkos types `Kokkos::View<int**>` as well as `int**` so that node/element ID calculations can be performed in GPU and CPU. Class `MeshOffsets` contains the auxilliary data structures discussed here.

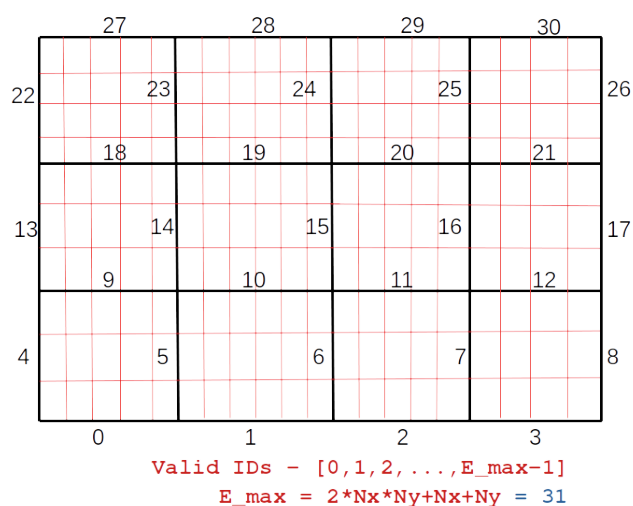
4.3 Block Entity Classification

Every submesh block in a 2D mesh is characterized by 4 edges and 4 vertices



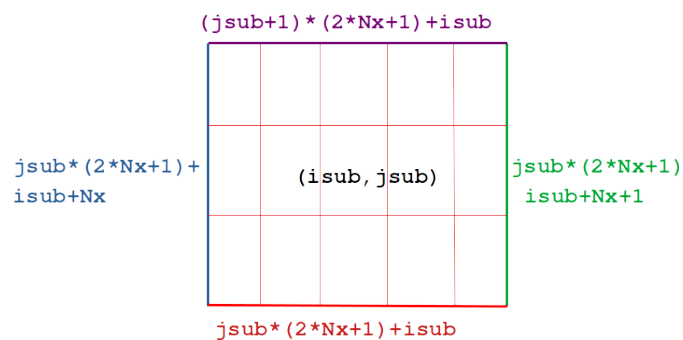
A vertex is a *zero-dimensional entity* while the edge is an *one-dimensional entity*. In order to apply different boundary/interface conditions at different locations, we need to ID these entities. In the following sections the convention used for edge and vertex numbering is explained. The convention is standard i.e. regardless of the activity of blocks in the mesh the convention holds.

Standard Block-Edge Numbering

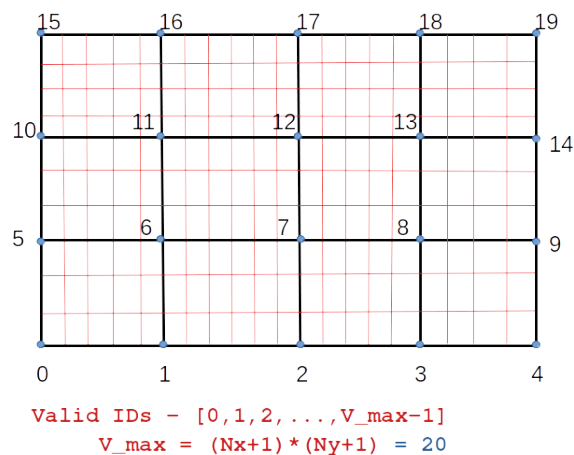


The standardized ID-ing helps us come up with expressions to obtain the edge IDs of any given block as shown below

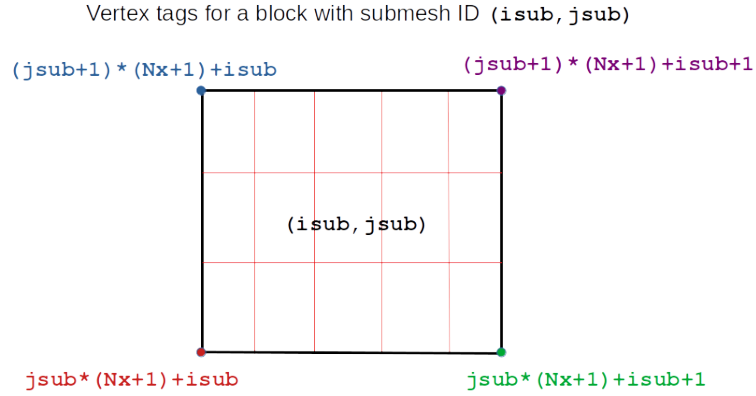
Edge tags for a block with submesh ID (i_{sub}, j_{sub})



Standard Block-Vertex Numbering

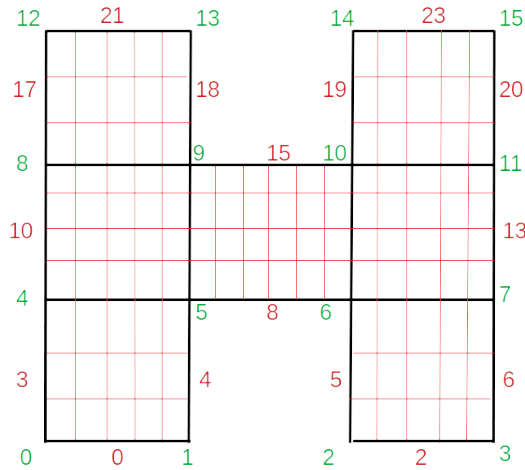


Similarly, for a given block the vertex IDs can be expressed as shown below



For a mesh with inactive blocks, the scheme remains unchanged i.e. the tags corresponding entities in non-active blocks will be invalid

Example – Boundary **Block-Edge-IDs** and **Block-Vertex-IDs**



4.4 Boundary Edge/Vertex Information

A boolean value for each edge/vertex stores if the given entity falls on the domain boundary or not. The library provides the API `where_is_node()` which takes the component-wise node IDs (`knode_x1`, `knode_x2`) and returns necessary information on the node such as if node is active, if node is on boundary, if a boundary node is on block-edge or block-vertex. Refer to later sections for API usage. Aside from boundary classification of block edges/vertices, the boundary normal directions are also stored on all boundary entities.

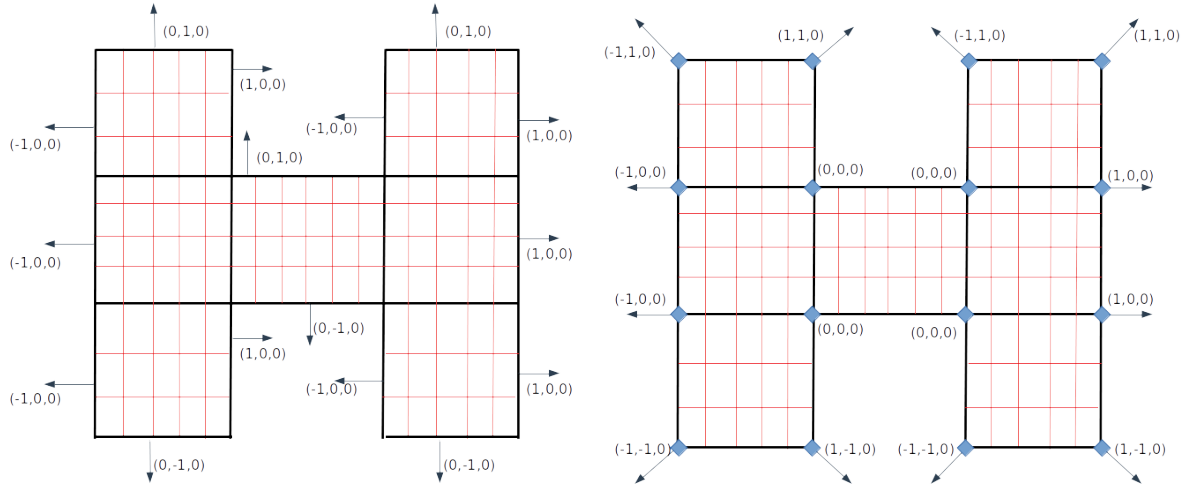
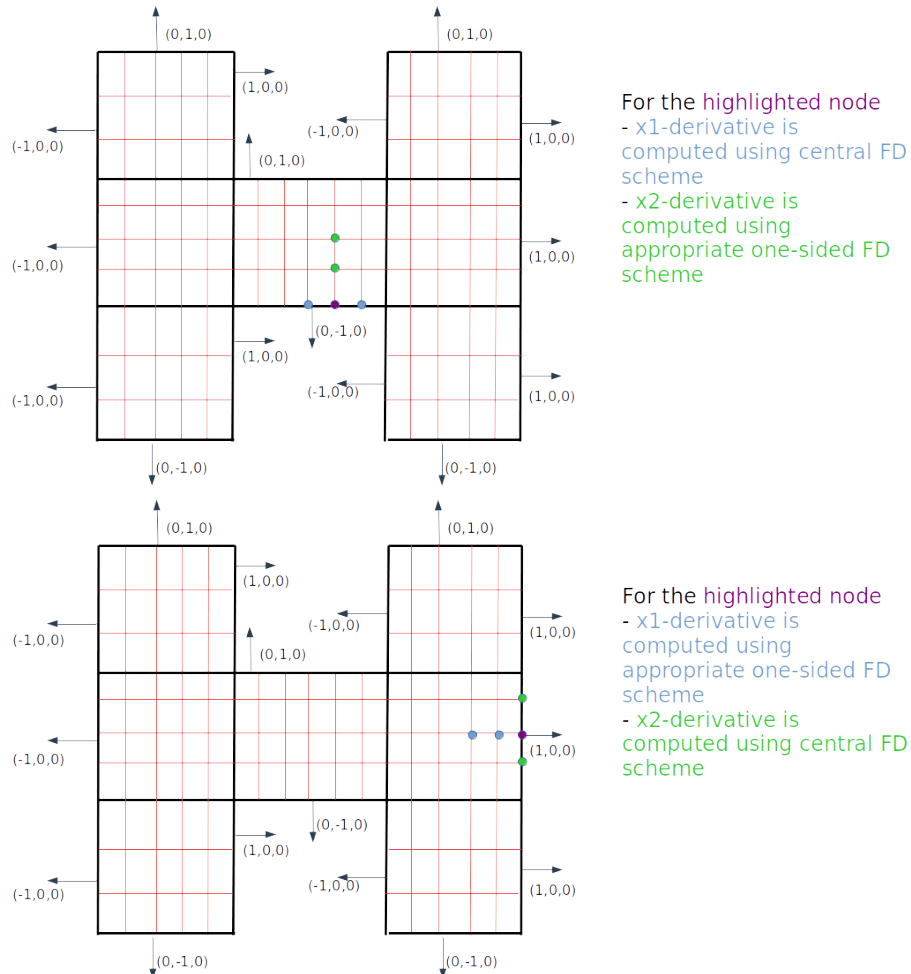


Figure 13: Boundary edge normals (left) and boundary vertex normals (right)

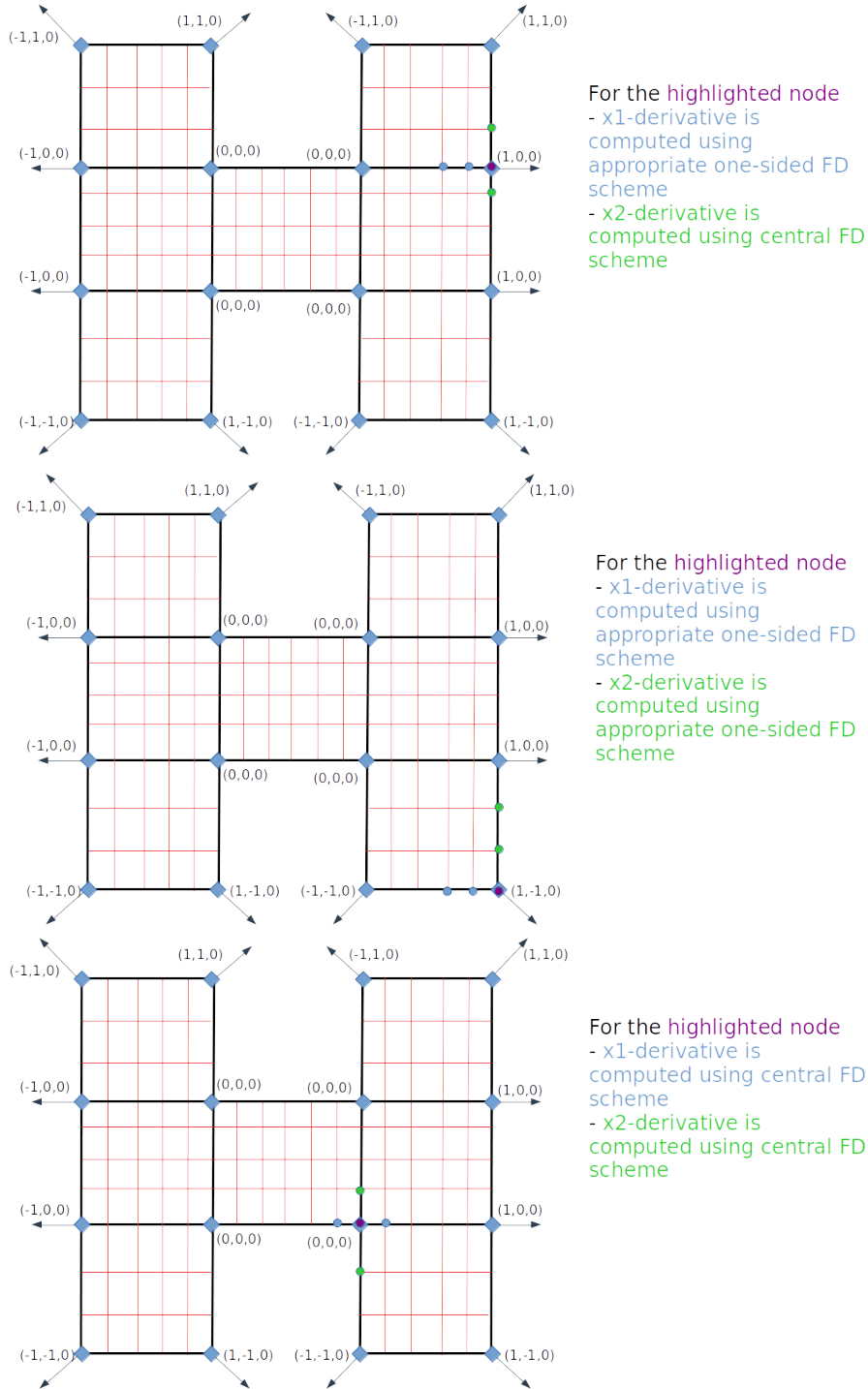
4.5 Gradient computation on boundaries

The normal directions stored on each block-boundary entity defines the rule for gradient computation. The normal direction determines the correct one-sided finite difference (FD) scheme to be implemented. (For second-order accurate gradient, fields from two adjacent nodes are required)

See below, for examples on gradient computation at nodes that falls on boundary block-edge



See below, for examples on gradient computation at nodes that falls on boundary block-vertices



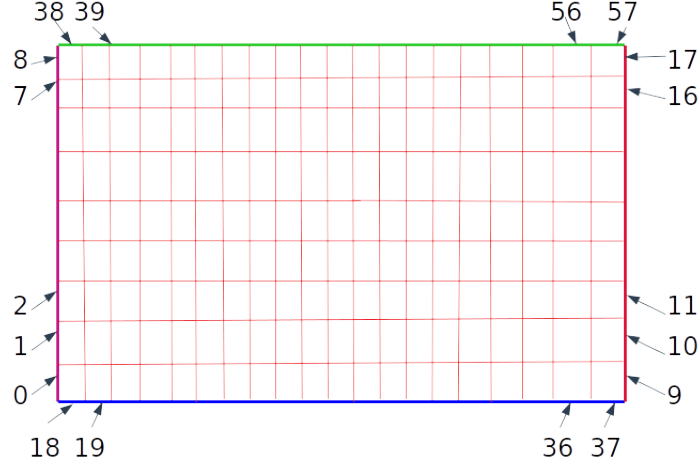
Notice, in the third example, while the node is on boundary it can be treated as a interior node for gradient calculation i.e. central FD scheme can be applied. Hence why the boundary normal for such nodes are set to a zero vector.

4.6 Boundary Element Face IDs

Boundary faces are defined by the part of a mesh element that touches the boundary of a domain. In 1D, a boundary face is a zero-dimensional entity i.e. point. In 2D, it is a one-dimensional entity i.e. line.

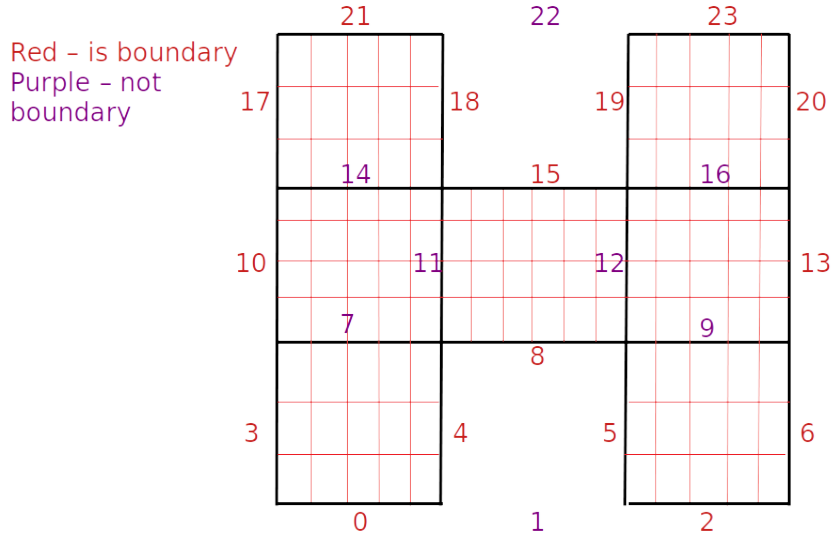
In **hpic2**, there are four possible boundary IDs – east, west, north, south. The boundary element faces are numbered on each boundaries as shown below. First west boundary faces are numbered, then east, then south and finally north faces.

Possible boundary ID in full uniform meshes are **east**, **west**, **north**, **south**



In **pumiMBBL**, the possible boundary IDs are set of block-edge tags which are classified as boundary. See example below,

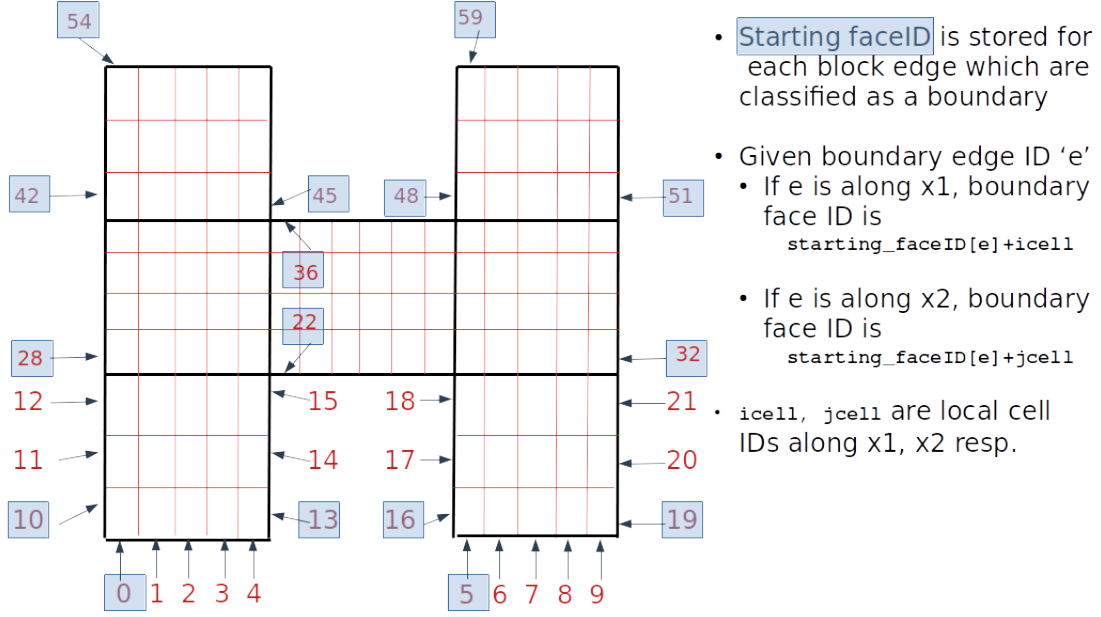
BDRY-IDs = {0,2,3,4,5,6,10,13,15,17,18,19,20,21,23}



In **pumiMBBL**,

- Boundary element faces are numbered on each classified boundary edges
- For the example with $BDRY-IDs = \{0, 2, 3, 4, 5, 6, 10, 13, 15, 17, 18, 19, 20, 21, 23\}$, first element faces in block-edge #0 are numbered, then block-edge #2, then #3 and so on till block-edge #23
- Numbering order inside an block-edge is done towards the positive direction
 - For block-edges along x1 – faces are numbered from left to right
 - For block-edges along x2 – faces are numbered from bottom to top

- The starting face ID on each edge is stored, to quickly fetch the face ID of a give a boundary edge ID and an element on that boundary



5 2D weight calculations from 1D linear weights

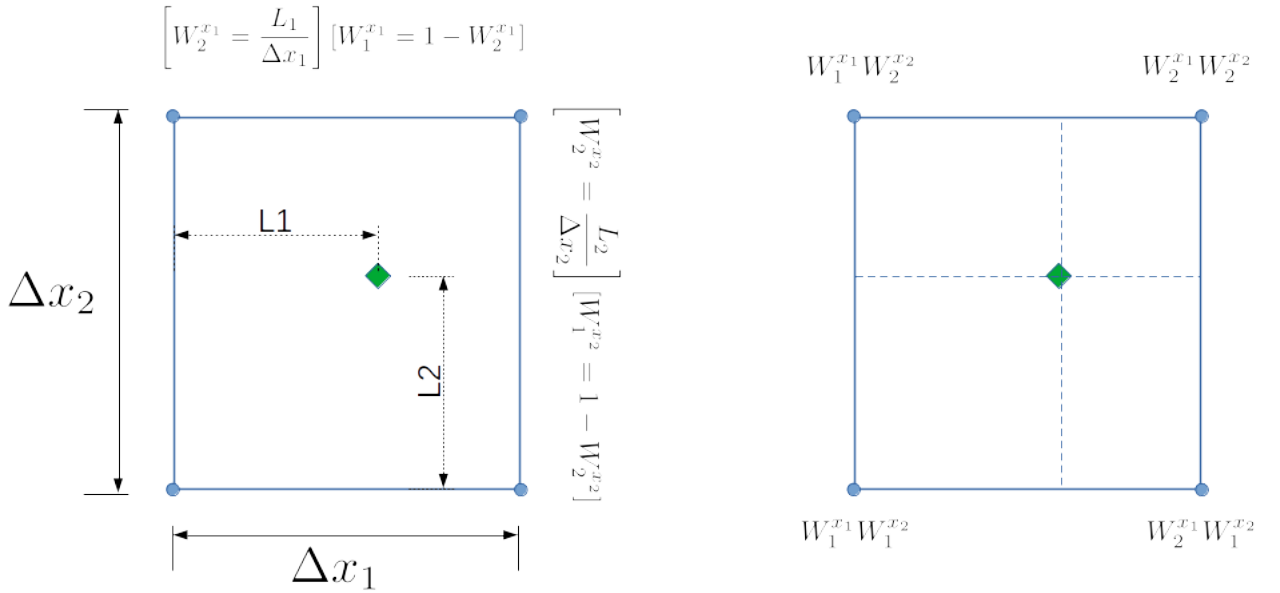


Figure 14: (Left) 1D component-wise weight computation
(Right) 1D component-wise weights to 2D weights

6 pumiMBBL-APIs – Get Information (ON HOST)

Mesh-Level Info:

API Name	Args	Return Type	Description
<code>get_global_x1_min</code>	MBBL pumi_obj	double	Domain lower bound x1-coordinate
<code>get_global_x2_min</code>	MBBL pumi_obj	double	Domain lower bound x2-coordinate
<code>get_global_x1_max</code>	MBBL pumi_obj	double	Domain upper bound x1-coordinate
<code>get_global_x2_max</code>	MBBL pumi_obj	double	Domain upper bound x2-coordinate
<code>get_total_mesh_elements</code>	MBBL pumi_obj	int	Total active elements in mesh
<code>get_total_mesh_nodes</code>	MBBL pumi_obj	int	Total active nodes in mesh
<code>get_total_x1_elements</code>	MBBL pumi_obj	int	Total elements along x1
<code>get_total_x2_elements</code>	MBBL pumi_obj	int	Total elements along x2
<code>get_mesh_volume</code>	MBBL pumi_obj	double	Total length in 1D, Active area in 2D of domain

Submesh-Level Info:

API Name	Args	Return Type	Description
<code>get_num_x1_submesh</code>	MBBL pumi_obj	int	Number of blocks along x1
<code>get_num_x2_submesh</code>	MBBL pumi_obj	int	Number of blocks along x2
<code>get_total_submesh_blocks</code>	MBBL pumi_obj	int	Total mesh blocks in domain including inactive blocks
<code>get_num_x1_elems_in_submesh</code>	MBBL pumi_obj, int isub	int	Number of elements along x1 in given block
<code>get_num_x2_elems_in_submesh</code>	MBBL pumi_obj, int jsub	int	Number of elements along x2 in given block
<code>get_num_x1_elems_before_submesh</code>	MBBL pumi_obj, int isub	int	Number of elements along x1 in all preceding blocks
<code>get_num_x2_elems_before_submesh</code>	MBBL pumi_obj, int jsub	int	Number of elements along x2 in all preceding blocks
<code>get_total_elements_in_block</code>	MBBL pumi_obj, int isub, int jsub	int	Total elements in given submesh block
<code>get_total_elements_in_block</code>	MBBL pumi_obj, int flattened_sub_ID	int	Total elements in given submesh block
<code>is_block_active_host</code>	MBBL pumi_obj, int isub, int jsub	bool	Activity status of given submesh block
<code>is_block_active_host</code>	MBBL pumi_obj, int flattened_sub_ID	bool	Activity status of given submesh block

Submesh Vertex related Info:

API Name	Args	Return Type	Description
<code>get_total_mesh_block_verts</code>	MBBL pumi_obj	int	Total number of block-vertices in mesh
<code>is_vert_bdry</code>	MBBL pumi_obj, int vertID	bool	Is given vertex ID part of domain boundary
<code>get_bdry_vert_normal</code>	MBBL pumi_obj, int vertID	Vector3	Boundary normal direction for given vertex ID

Submesh Edge related Info:

API Name	Args	Return Type	Description
<code>get_total_mesh_block_edges</code>	MBBL pumi_obj	int	Total number of block-edges in mesh
<code>is_edge_bdry</code>	MBBL pumi_obj, int edgeID	bool	Is given edge ID part of domain boundary
<code>get_bdry_edge_normal</code>	MBBL pumi_obj, int edgeID	Vector3	Boundary normal direction for given edge ID
<code>get_west_edgeID</code>	MBBL pumi_obj, int isub, int jsub	int	edge ID of west edge of given submesh block
<code>get_east_edgeID</code>	MBBL pumi_obj, int isub, int jsub	int	edge ID of east edge of given submesh block
<code>get_south_edgeID</code>	MBBL pumi_obj, int isub, int jsub	int	edge ID of south edge of given submesh block
<code>get_north_edgeID</code>	MBBL pumi_obj, int isub, int jsub	int	edge ID of north edge of given submesh block
<code>get_num_faces_on_edge</code>	MBBL pumi_obj, int edgeID	int	Number of elements along a given edge ID
<code>get_starting_faceID_on_bdry_edge</code>	MBBL pumi_obj, int edgeID	int	Starting boundary element face ID on give boundary edge

Mesh Element/Node Info

API Name	Args	Return Type	Description
<code>get_node_submeshID</code>	MBBL pumi_obj, int knode_x1, int knode_x2	int	Returns the active flattened submesh ID to which a given node belongs to
<code>get_elem_submeshID</code>	MBBL pumi_obj, int kcell_x1, int kcell_x2	int	Returns the active flattened submesh ID to which a given element belongs to
<code>get_global_nodeID_2D</code>	MBBL pumi_obj, int knode_x1, int knode_x2	int	Returns the global node ID (on a 2D mesh) for a given node
<code>get_global_elemID_2D</code>	MBBL pumi_obj, int kcell_x1, int kcell_x2	int	Returns the global element ID (on a 2D mesh) for a given element
<code>where_is_node</code>	MBBL pumi_obj, int knode_x1, int knode_x2, bool *on_bdry, bool *in_domain, int *bdry_tag, int *bdry_dim	void	For given node, tells if - if node is on domain boundary - if node is in active domain - boundary tag (if node is on boundary) - boundary dimension (0=block-vertex, 1=block-edge – if node is on boundary)

Directional and flattened IDs

API Name	Args	Return Type	Description
<code>flatten_submeshID_and_cellID_host</code>	MBBL pumi_obj, int isub, int icell, int jsub, int jcell, int *flattened_subID, int *flattened_cellID	void	Converts the directional submesh and cell IDs to flattened submesh and cell IDs on a 2D mesh
<code>get_directional_submeshID_and_cellID_host</code>	MBBL pumi_obj, int flattened_subID, int flattened_cellID, int *isub, int *icell, int *jsub, int *jcell,	void	Converts the flattened submesh and cell IDs to directional submesh and cell IDs on a 2D mesh

7 pumiMBBL-APIs – Particle Operations (ON HOST)

API Name	Args	Return Type	Description
<code>get_rand_point_in_mesh_host</code>	MBBL pumi_obj	Vector3	Generates a random point coordinate in active region of the domain
<code>is_point_in_mesh_host</code>	MBBL pumi_obj, Vector3 coords	bool	Tells if a give point coordinates is in active domain
<code>locate_submesh_and_cell_x1_host</code>	MBBL pumi_obj, double x1.coord int *isub, int *icell	void	Computes a give x1-coordinate's submesh ID and cell ID (along x1-direction) Performs global search
<code>locate_submesh_and_cell_x2_host</code>	MBBL pumi_obj, double x2.coord int *jsub, int *jcell	void	Computes a give x2-coordinate's submesh ID and cell ID (along x2-direction) Performs global search
<code>calc_weights_x1_host</code>	MBBL pumi_obj, double x1.coord int isub, int icell, int *kcell_x1, double *Wgh2_x1	void	Computes linear weight (along x1 direction) for the max-side node as well as component-wise global cell ID (along x1), given x1-coords, submesh ID and cell ID
<code>calc_weights_x2_host</code>	MBBL pumi_obj, double x2.coord int jsub, int jcell, int *kcell_x2, double *Wgh2_x2	void	Computes linear weight (along x2 direction) for the max-side node as well as component-wise global cell ID (along x2), given x2-coords, submesh ID and cell ID
<code>calc_global_cellID_and_nodeID_host</code>	MBBL pumi_obj, int isub, int jsub, int kcell_x1, int kcell_x2, int *global_cellID, int *inp1, int *inp3	void	Computes global cell and node IDs for a given element (NEEDS the component-wise submesh and cell IDs of the element). Node IDs of bottom-left and top-left nodes of the element are computed. ONLY for 2D mesh

8 pumiMBBL APIs – ON DEVICE

Submesh-Level Info

API Name	Args	Return Type	Description
<code>get_x1_cellID</code>	MBBL pumi_obj, int isub, int icell	int	Global cell ID along x1
<code>get_x2_cellID</code>	MBBL pumi_obj, int jsub, int jcell	int	Global cell ID along x2
<code>get_x1_elem_size_in_submesh</code>	MBBL pumi_obj, int isub, int icell	double	Element size along x1
<code>get_x2_elem_size_in_submesh</code>	MBBL pumi_obj, int jsub, int jcell	double	Element size along x2

Directional and flattened IDs

API Name	Args	Return Type	Description
<code>flatten_submeshID_and_cellID</code>	MBBL pumi_obj, int isub, int icell, int jsub, int jcell, int *flattened_subID, int *flattened_cellID	void	Converts the directional submesh and cell IDs to flattened submesh and cell IDs on a 2D mesh
<code>get_directional_submeshID_and_cellID</code>	MBBL pumi_obj, int flattened_subID, int flattened_cellID, int *isub, int *icell, int *jsub, int *jcell,	void	Converts the flattened submesh and cell IDs to directional submesh and cell IDs on a 2D mesh

Particle Operations

API Name	Args	Return Type	Description
locate_submesh_and_cell_x1	MBBL pumi_obj, double x1_coord int *isub, int *icell	void	Computes a give x1-coordinate's submesh ID and cell ID (along x1-direction) Performs global search
locate_submesh_and_cell_x2	MBBL pumi_obj, double x2_coord int *jsub, int *jcell	void	Computes a give x2-coordinate's submesh ID and cell ID (along x2-direction) Performs global search
update_submesh_and_cell_x1	MBBL pumi_obj, double x1_coord int isub_old, int icell_old, int *isub, int *icell	void	Computes a give x1-coordinate's submesh ID and cell ID (along x1-direction) Performs adjacency search with old IDs
update_submesh_and_cell_x2	MBBL pumi_obj, double x2_coord int jsub_old, int jcell_old, int *jsub, int *jcell	void	Computes a give x2-coordinate's submesh ID and cell ID (along x2-direction) Performs adjacency search with old IDs
calc_weights_x1	MBBL pumi_obj, double x1_coord int isub, int icell, int *kcell_x1, double *Wgh2_x1	void	Computes linear weight (along x1 direction) for the max-side node as well as component-wise global cell ID (along x1), given x1-coords, submesh ID and cell ID
calc_weights_x2	MBBL pumi_obj, double x2_coord int jsub, int jcell, int *kcell_x2, double *Wgh2_x2	void	Computes linear weight (along x2 direction) for the max-side node as well as component-wise global cell ID (along x2), given x2-coords, submesh ID and cell ID
calc_global_cellID_and_nodeID	MBBL pumi_obj, int isub, int jsub, int kcell_x1, int kcell_x2, int *global_cellID, int *inp1, int *inp3	void	Computes global cell and node IDs for a given element (NEEDS the component-wise submesh and cell IDs of the element). Node IDs of bottom-left and top-left nodes of the element are computed. ONLY for 2D mesh
push_particle	MBBL pumi_obj, Vector3 q, Vector3 dq, int *isub, int *jsub, int *icell, int *jcell, bool *crossed_bdry, int *bdry_edge_hit, double *fraction_done, int *bdry_faceID_crossed	Vector3	Particle push performed with convex tracking. Input particle coordinate vector, displacement vector and IDs. Output info includes - updated submesh/cell IDs with adjacency search - did particle cross boundary - boundary ID crossed by particle - fraction of push completed - boundary face ID crossed

9 PUMI-API Usage

9.1 Particle-related APIs

To locate a particle (and compute its partial nodal weight contributions to the 4 nodes) whose coordinates are (q_1, q_2) in 2D,

Step-1 Locate the submesh ID and local cell ID (in each direction)

Step-2 Get the partial component-wise weights corresponding to the max-size node and component-wise global cell IDs (in each direction)

Step-3 Get the global cell and node IDs in 2D

Step-4 Compute remaining weights and distribute to the corresponding nodes

```
int isub, jsub, icell, jcell, kcell_x1, kcell_x2;
int global_2D_cell, left_bottom_node, left_top_node;
double Wgh2_x1, Wgh2_x2, Wgh1_x1, Wgh1_x2;

// Step-1
pumi::locate_submesh_and_cell_x1(pumi_obj, q1, &isub, &icell);
pumi::locate_submesh_and_cell_x2(pumi_obj, q2, &jsub, &jcell);
// After thist step variables isub, jsub will contain component-wise submesh IDs
// and icell, jcell will contain local cell IDs

//Step-2
pumi::calc_weights_x1(pumi_obj, q1, isub, icell, &kcell_x1, &Wgh2_x1);
pumi::calc_weights_x2(pumi_obj, q2, jsub, jcell, &kcell_x2, &Wgh2_x2);
// After thist step variables kcell_x1, kcell_x2 will contain component-wise global
cell IDs and Wgh2_x1, Wgh2_x2 will contain component-wise weights

//Step-3
pumi::calc_global_cellID_and_nodeID(pumi_obj, kcell_x1, kcell_x2,
&global_2D_cell, &left_bottom_node, &left_top_node);
// After thist step variables global_2D_cell will contain the global element ID and
left_bottom_node, left_top_node will contain the relevant global node IDs

//Step-4
Wgh1_x1 = 1.0 - Wgh2_x1;
Wgh1_x2 = 1.0 - Wgh2_x2;
int right_bottom_node = left_bottom_node + 1;
int right_top_node = left_top_node + 1;
//distribute to relevant nodes
density[left_bottom_node] += Wgh1_x1*Wgh1_x2;
density[left_top_node]    += Wgh1_x1*Wgh2_x2;
density[right_bottom_node] += Wgh2_x1*Wgh1_x2;
density[right_top_node]   += Wgh2_x1*Wgh2_x2;
```

For a pushed particle whose previous submesh and cell IDs are known use the update routines (implements adjacency search with stored BL node coordinates)

```

int isub, jsub, icell, jcell, kcell_x1, kcell_x2;
int global_2D_cell, left_bottom_node, left_top_node;
double Wgh2_x1, Wgh2_x2;

// Step-1
pumi::update_submesh_and_cell_x1(pumi_obj, q1, isub, icell, &isub, &icell);
pumi::update_submesh_and_cell_x2(pumi_obj, q2, jsub, jcell, &jsub, &jcell);
// After this step variables isub, jsub will contain component-wise submesh IDs
// and icell, jcell will contain local cell IDs

//Step-2 to Step-4 remains unchanged

```

9.2 Particle Push (with inactive blocks)

In case of a domain with inactive blocks particle's coordinate update, previously performed as,

```

q1 += dq1;
q2 += dq2;
pumi::update_submesh_and_cell_x1(pumi_obj, q1, isub, icell, &isub, &icell);
pumi::update_submesh_and_cell_x2(pumi_obj, q2, jsub, jcell, &jsub, &jcell);

```

needs to be replaced with

```

pumi::push_particle(pumi_obj, q1, q2, dq1, dq2, &isub, &jsub, &icell, &jcell,
                  &in_domain, &bdry_hit);

```

In addition to updating the submesh and cell IDs, we also update the boolean `in_domain` to `false` when a particle goes out of a boundary and the integer `bdry_hit` to tag of the edge which the particle crosses to go out of the boundary. For particles inside the domain, `in_domain` is set to `true` and `bdry_hit` is set to `-1`.

NOTE: All the above functions can be called in GPU (i.e. inside `Kokkos::parallel_for`). Check `pumiMBBL_test.cpp` for an example.

9.3 Field-related APIs

- The grading ratio (along a given direction) about a node (from its component-wise global node ID i.e. `i1`, `i2`) can be obtained as

```

// In x1-direction
double r1 = pumi::return_gradingratio(pumi_obj, pumi::x1_dir, i1);

// In x2-direction
double r2 = pumi::return_gradingratio(pumi_obj, pumi::x2_dir, i2);

```

- The element-length (along a given direction) for an element (from its component-wise global element/node ID i.e. `i1`, `i2`) can be obtained as

```

// In x1-direction
double e1 = pumi::return_elemsize(pumi_obj, pumi::x1_dir, kcell_x1,
pumi::elem_input_offset);

```

```
// In x2-direction
double e2 = pumi::return_elemsize(pumi_obj, pumi::x2_dir, kcell_x2,
pumi::elem_input_offset);
```

In the above example `kcell_x1`, `kcell_x2` are element IDs hence the fourth argument is an offset corresponding to element index inputs. If node IDs are to be provided as inputs i.e. `knode_x1`, `knode_x2`, then provide `pumi::elem_on_min_side_offset` for element on min-side of node and `pumi::elem_on_max_side_offset` for max-side element as last argument

- The nodal co-volume about a node (from its component-wise global node IDs i.e. `knode_x1`, `knode_x2`) can be obtained as

```
// In 1D mesh
double cv = pumi::return_covolume(pumi_obj, knode_x1);
// In 2D mesh
double cv = pumi::return_covolume(pumi_obj, knode_x1, knode_x2);
```

- Information such as active node, boundary node and entity tag/dimension of a boundary node can be obtained by the following function

```
bool in_domain, on_bdry;
int bdry_tag, bdry_dim;
pumi::where_is_node(pumi_obj, knode_x1, knode_x2,
                    &in_domain, &on_bdry, &bdry_tag, &bdry_dim);
```

After the API call

- If the input node is inside the domain (i.e. not in an inactive block), then `in_domain` will be set to `true`. Otherwise it will be `false`
- If the input node is a boundary node, then `on_bdry` will be set to `true`. Otherwise it will be `false`
- If the input node is a boundary node, then `bdry_tag` will be set to the tag of the entity to which the node belongs to. For inactive nodes, the tag will be set to `-999` and for non-boundary active nodes it will be `-1`
- If the input node is a boundary node, then `bdry_dim` will be set to `0` if it falls on block-vertex and `1` if it falls on block-edge. For non-boundary and inactive nodes it will be `-1`

NOTE: The above APIs can be called only from host. Check `pumiMBBL_test.cpp` for an example