pumiMBBL-GPU with Kokkos

Vignesh Vittal-Srinivasaragavan
RPI-SCOREC

June 10, 2021

1 New PUMI Data structure

1.1 submesh – base classes and derived classes

```
class Submesh{
public:
    double xmin;
    double xmax;
    int Nel;
    double t0;
    double r;
    int Nel_cumulative;
    pumi_meshtype meshtype; // enum type 0x01-uniform, 0x02-minBL and 0x04-maxBL
    Kokkos::View<double*> BL_coords;
    Submesh(....):....{}; //Constructor to submesh class

    virtual void locate_cell(); // using analtyical expressions
    virtual void update_cell(); // using adjacency search
    virtual void calc_weights(); // using stored node coordinates
};
```

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 class
    // uniform mesh APIs
    void locate_cell(); // using analtyical expressions
    void update_cell(); // using adjacency search
    void calc_weights();
    .
    .
    .
};
```

Left/Bottom BL meshing:

```
class MinBL_Submesh : public Submesh{
public:
    MinBL_Submesh(....):.....{}; // Constructor to minBL submesh class
    // minBL mesh APIs
    void locate_cell(); // using analtyical expressions
    void update_cell(); // using adjacency search
    void calc_weights();
    .
    .
    .
};
```

Right/Top BL meshing:

```
class MaxBL_Submesh : public pumi_submesh{
public:
    MaxBL_Submesh(....):.....{}; // Constructor to maxBL submesh class
    // maxBL mesh APIs
    void locate_cell(); // using analtyical expressions
    void update_cell(); // using adjacency search
    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` – class

```
class Mesh{
public:
    int ndim;
    int nsubmesh_x1;
    int nsubmesh_x2;
    Kokkos::View<bool**> isactive;
    int Nel_total_x1;
    int Nel_total_x2;
    .
    .
    Mesh(....):...{}; // Class constructor
    .
    .
    // will contain more inactive mesh realated members (for later)
};
```

Once again, this is similar to current `struct pumi_mesh` but objects to submesh arrays are no more members of mesh class

1.3 Typedefs for mesh/submesh objects

```
// 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*;

// Mesh object array allocated in GPU using Kokkos View
// can be copied to CPU using Kokkos::deep_copy()
using MeshDeviceViewPtr = Kokkos::View<Mesh*>;
```

1.4 Wrapper struct containing mesh and submesh objects

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

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

2 New 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 data structure
.
.
// Declare mesh object (in GPU)
pumi::MeshDeviceViewPtr mesh;

// Declare submesh objects (in GPU)
pumi::SubmeshDeviceViewPtr submesh_x1;
pumi::SubmeshDeviceViewPtr submesh_x2;

// Declare copy of submesh objects in CPU
pumi::SubmeshHostViewPtr host_submesh_x1;
pumi::SubmeshHostViewPtr host_submesh_x2;
```

```
// Set options for mesh
pumi::Mesh_Options pumi_options;
pumi_options.BL_storage_option = pumi::store_BL_coords_ON; //_OFF will need
//different API calls (not implemented yet)

// populate submesh objects (both GPU and CPU copy)
submesh_x1 =
pumi::submesh_initialize(pumi_inputs, pumi_options, pumi::x1_dir, &host_submesh_x1);

submesh_x2 =
pumi::submesh_initialize(pumi_inputs, pumi_options, pumi::x2_dir, &host_submesh_x2);

// populate mesh object
mesh = pumi::mesh_initialize(pumi_inputs,
submesh_x1, host_submesh_x1, submesh_x2, host_submesh_x2);

// Contruct the final pumi wrapper struct
pumi::MBBL pumi_obj(mesh, submesh_x1, host_submesh_x1, submesh_x2, host_submesh_x2);

// 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)`

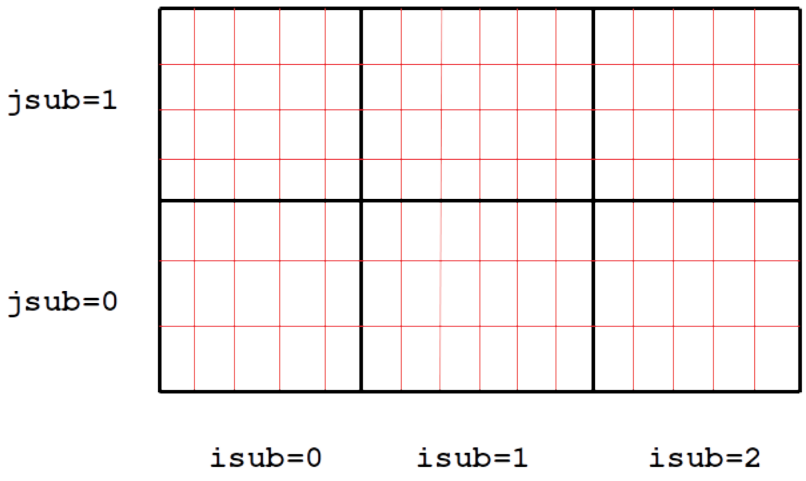


Figure 1: `isub` – submesh ID in x1-direction
`jsub` – submesh ID in x2-direction

Possible values:

$$0 \leq \texttt{isub} \leq N_{x_1}^{submesh} - 1 \qquad 0 \leq \texttt{jsub} \leq N_{x_2}^{submesh} - 1$$

$N_{x_1}^{submesh}, N_{x_2}^{submesh}$ – Number of submesh blocks in x1, x2 directions respectively

Local Cell IDs

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

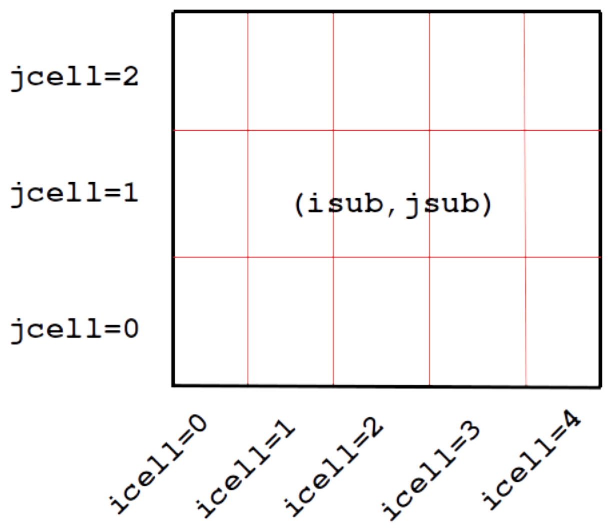


Figure 2: `icell` – local cell ID in x1-direction
`jcell` – local cell ID in x2-direction

Possible values:

$$0 \leq \texttt{icell} \leq N^{el}_{x_1,\texttt{isub}} - 1 \qquad 0 \leq \texttt{jcell} \leq N^{el}_{x_2,\texttt{jsub}} - 1$$

$N^{el}_{x_1,\texttt{isub}}$ – Number of elements along x1-direction in x1-block with ID `isub`
$N^{el}_{x_2,\texttt{jsub}}$ – Number of elements along x2-direction in x2-block with ID `jsub`

Component-wise Global Cell IDs

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

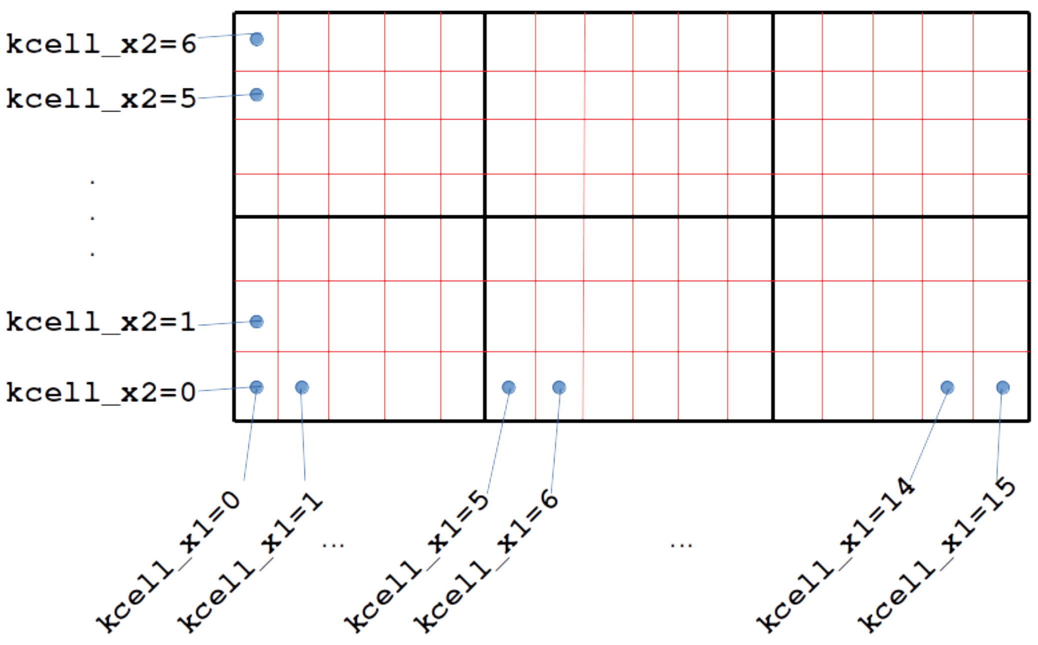


Figure 3: `kcell_x1` – component-wise global cell ID in x1-direction
`kcell_x2` – component-wise global cell ID in x2-direction

Possible values:

$$0 \leq \texttt{kcell\_x1} \leq N_{x_1}^{el,tot} - 1 \qquad 0 \leq \texttt{kcell\_x2} \leq N_{x_2}^{el,tot} - 1$$

$N_{x_1}^{el,tot}, N_{x_2}^{el,tot}$ – Number of total elements along x1, x2 directions respectively

Component-wise Global Node IDs

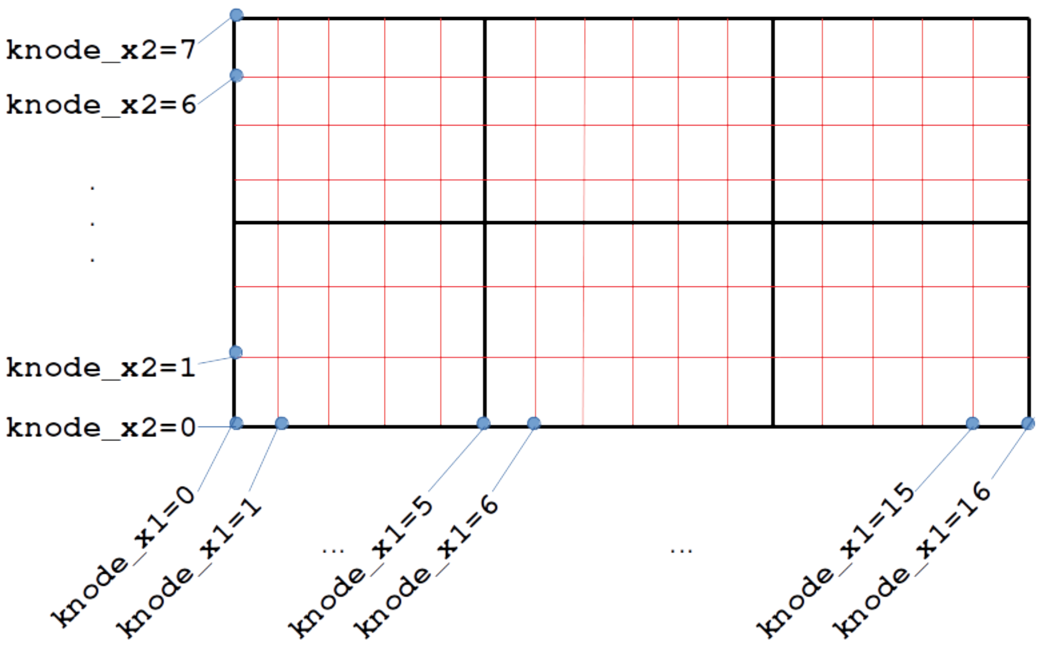


Figure 4: `knode_x1` – component-wise global cell ID in x1-direction
`knode_x2` – component-wise global cell ID in x2-direction

Possible values:

$$0 \leq \texttt{knode\_x1} \leq N_{x_1}^{el,tot} \qquad 0 \leq \texttt{knode\_x2} \leq N_{x_2}^{el,tot}$$

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

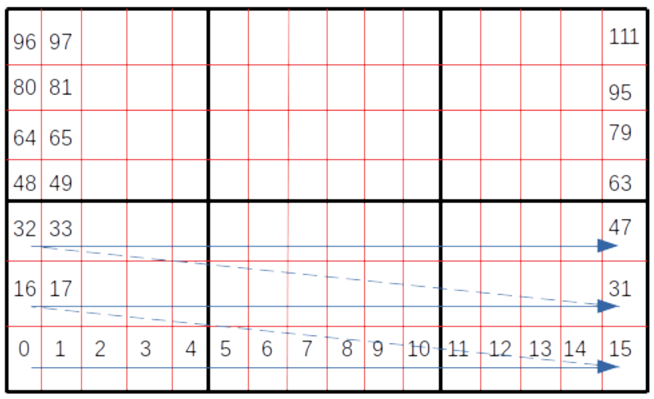


Figure 5: $N_{x_1}^{submesh} = 3, N_{x_1}^{el,tot} = 16$ $N_{x_2}^{submesh} = 2, N_{x_2}^{el,tot} = 7$

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

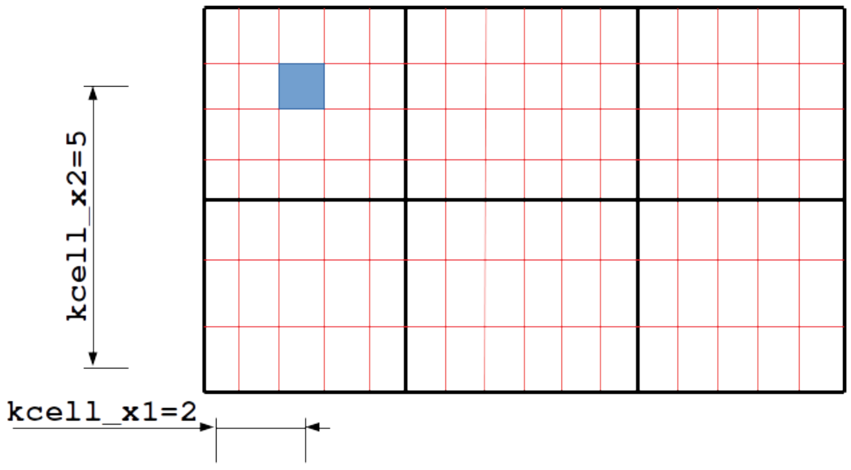


Figure 6: $\texttt{global\_2D\_cell} = \texttt{kcell\_x2} \times N_{x_1}^{el,tot} + \texttt{kcell\_x1} = 5(16) + 2 = 82$

Possible values:

$$0 \leq \texttt{global\_2D\_cell} \leq N_{2D}^{el,tot} - 1$$

$N_{2D}^{el,tot}$ – 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

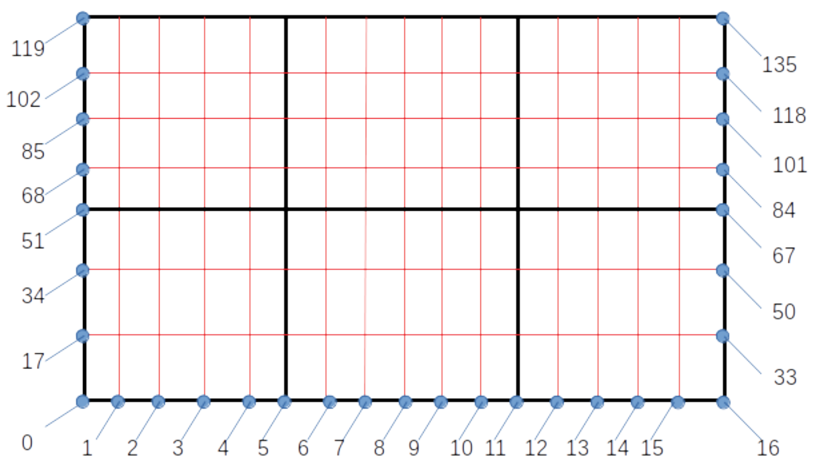


Figure 7: Node numbering is done similar to element numbering

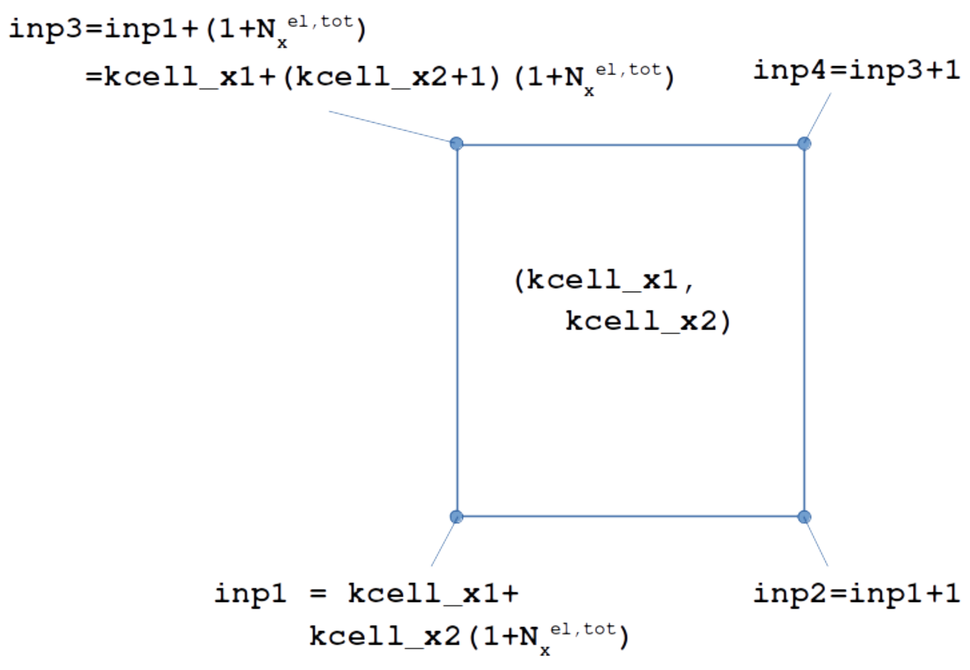


Figure 8: Global Node IDs associated with a given element can be calculated from global column index `kcell_x1`, global row index `kcell_x2` and total elements along x1-direction $N_{x_1}^{el,tot}$

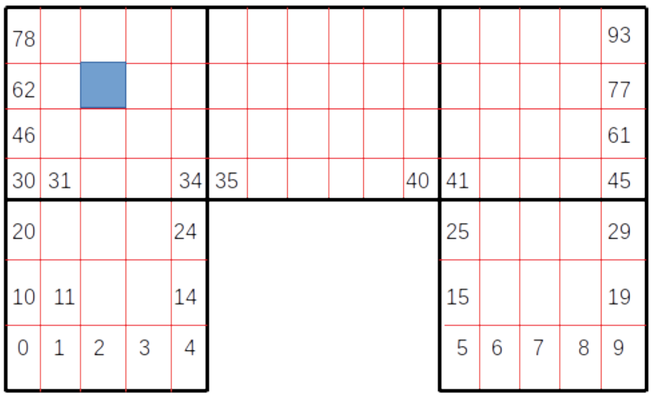
It's only essential to compute the node IDs of left-bottom `(inp1)` and left-top `(inp3)` nodes. Their right counterparts `(inp2 and inp4)` can be obtained by incrementing the values by 1.
Possible values:

$$0 \leq \texttt{inp} \leq N_{2D}^{np,tot} - 1$$

$N_{2D}^{np,tot}$ – Total number of nodes in the 2D domain.
On a full mesh $N_{2D}^{np,tot} = \left(N_{x_1}^{el,tot} + 1\right) \times \left(N_{x_2}^{el,tot} + 1\right)$

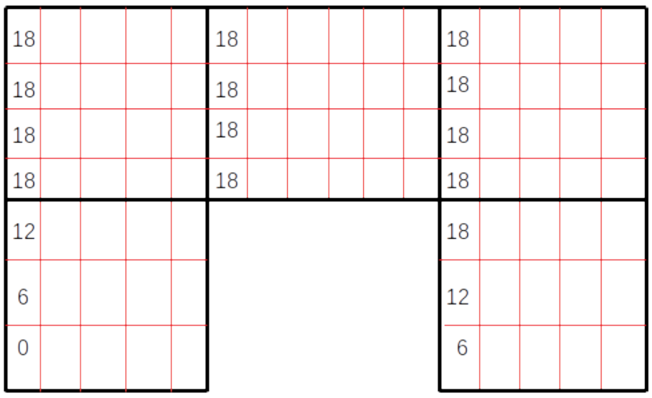
4.2 Mesh with Inactive Blocks

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



Elements in the inactive blocks are skipped while numbering. For the sake of analytical computation of node IDs it is necessary to keep track of how many elements are skipped for for each row in each x1-block (see figure below)

`elemoffset[isub][kcell_x2]`



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)

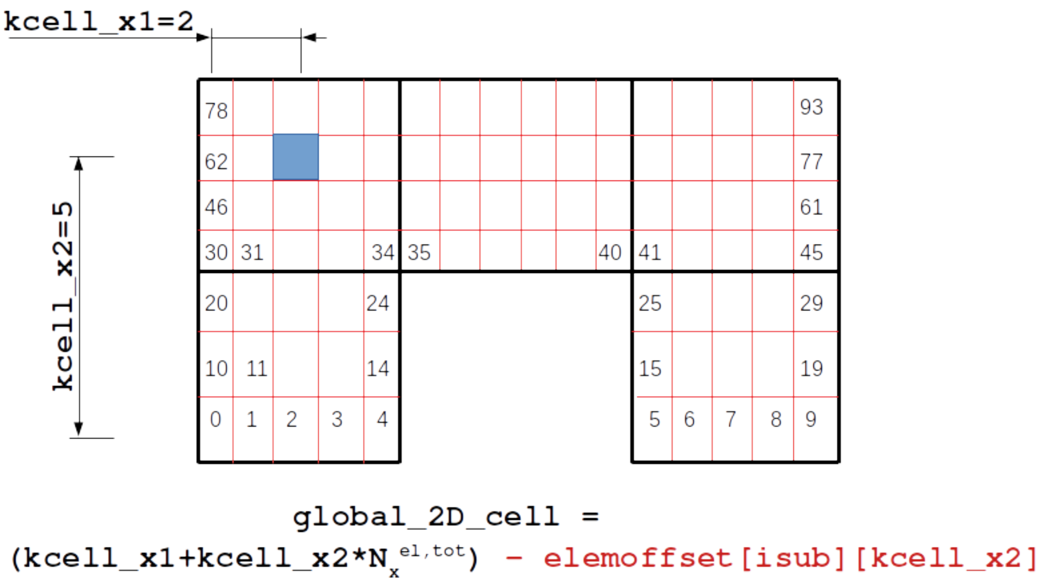
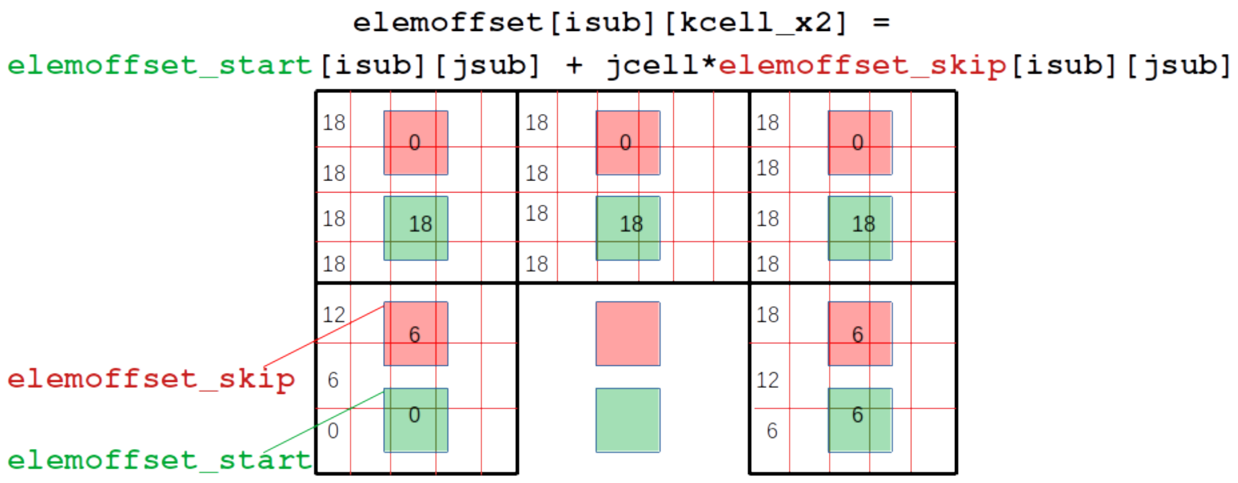
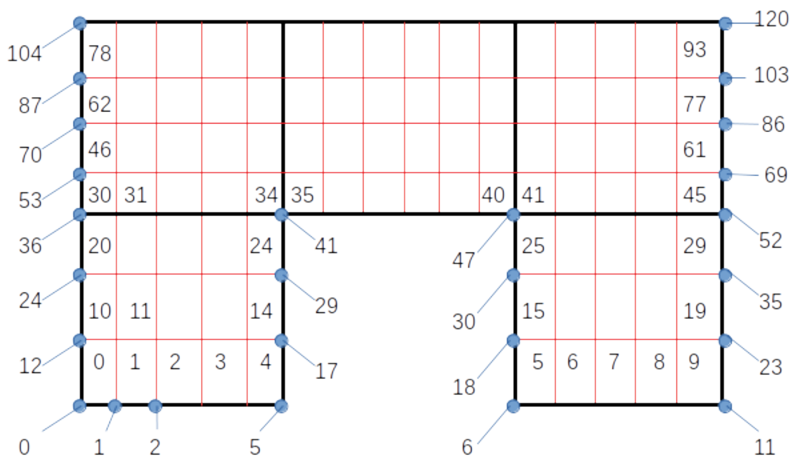


Figure 9: `global_2D_cell = 2+5×16-18 = 64`

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]
18 18 18 18
0
18
18 18 18 18
0
18
18 18 18 18
0
18
12 6 0
6
0
18 12 6
6
6
elemoffset_skip
elemoffset_start

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.

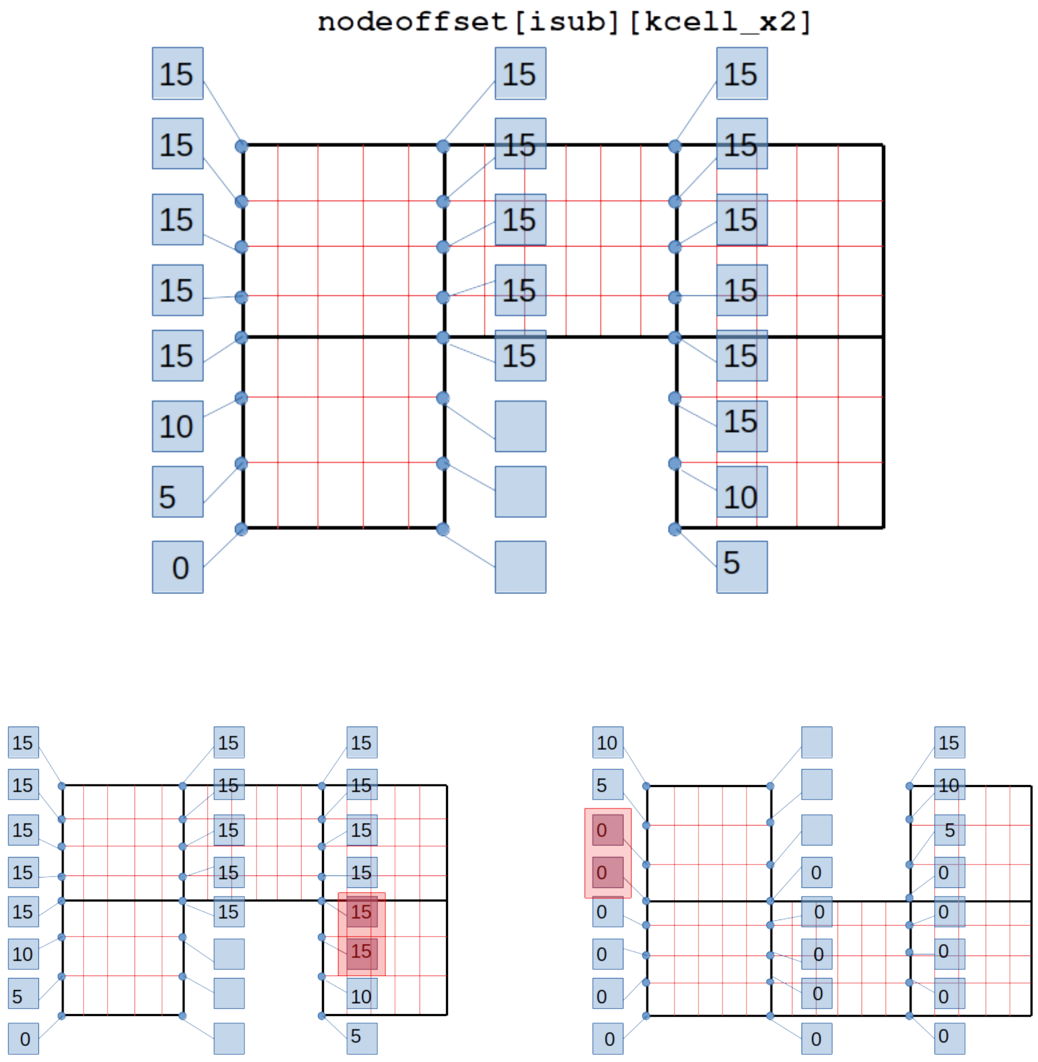
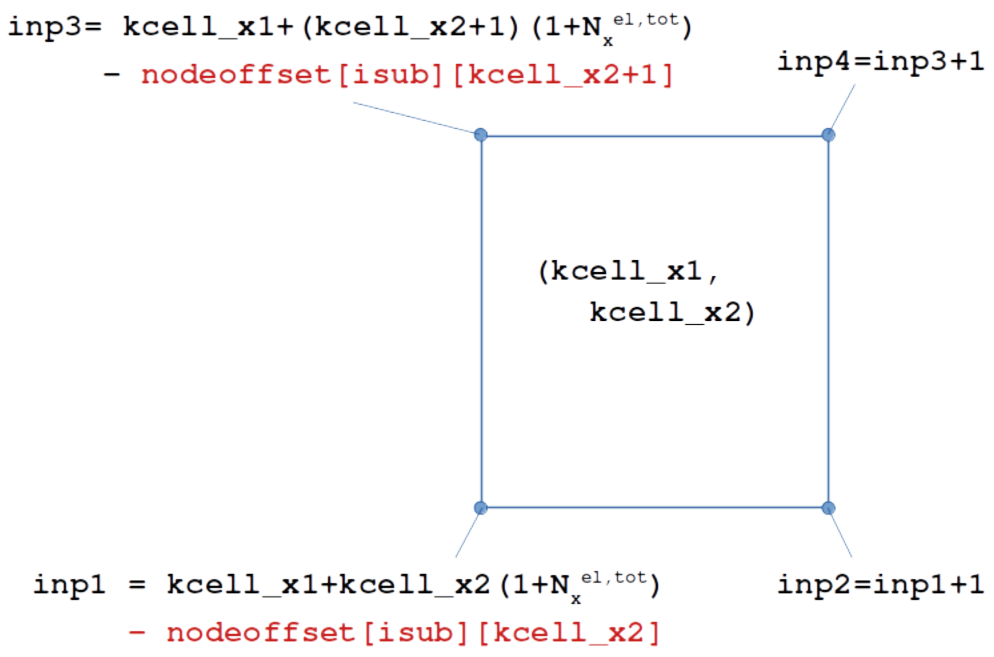


Figure 10: Examples where `nodeoffset` do not follow the same pattern as the rest of the block


inp3= kcell_x1+(kcell_x2+1)(1+N_x^{el,tot})
– nodeoffset[isub][kcell_x2+1]
inp4=inp3+1
(kcell_x1,
kcell_x2)
inp1 = kcell_x1+kcell_x2(1+N_x^{el,tot})
– nodeoffset[isub][kcell_x2]
inp2=inp1+1

Node ID calculation – Examples

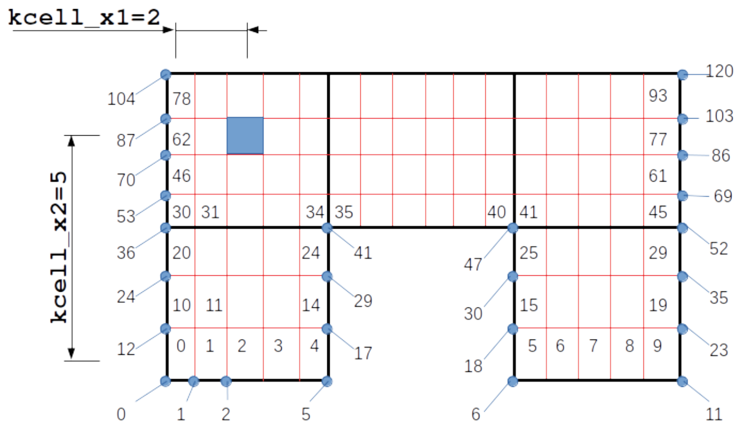


Figure 11: `inp1=2+5×17-15=72`
`inp3=2+6×17-15=89`

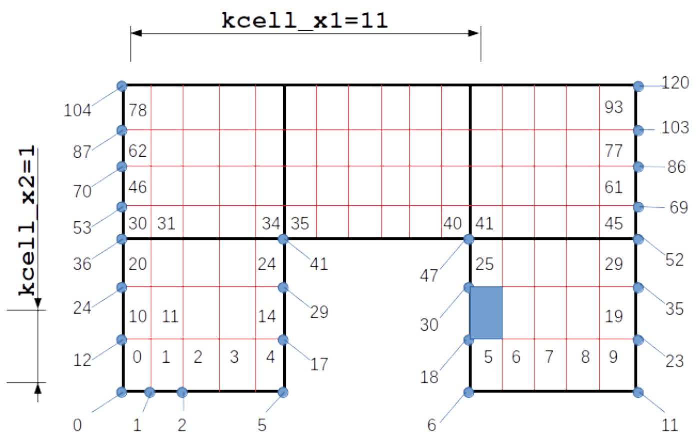
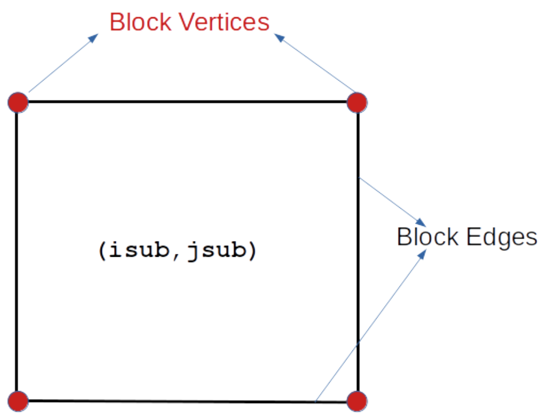


Figure 12: `inp1=11+1×17-10=18`
`inp3=11+2×17-15=30`

NOTE: The auxiliary data structures `nodeoffset, elemoffset_start, elemoffset_skip` are all allocated as kokkos types `Kokkos::View<double**>` so that node/element ID calculations can be performed in GPU in the particle loop.

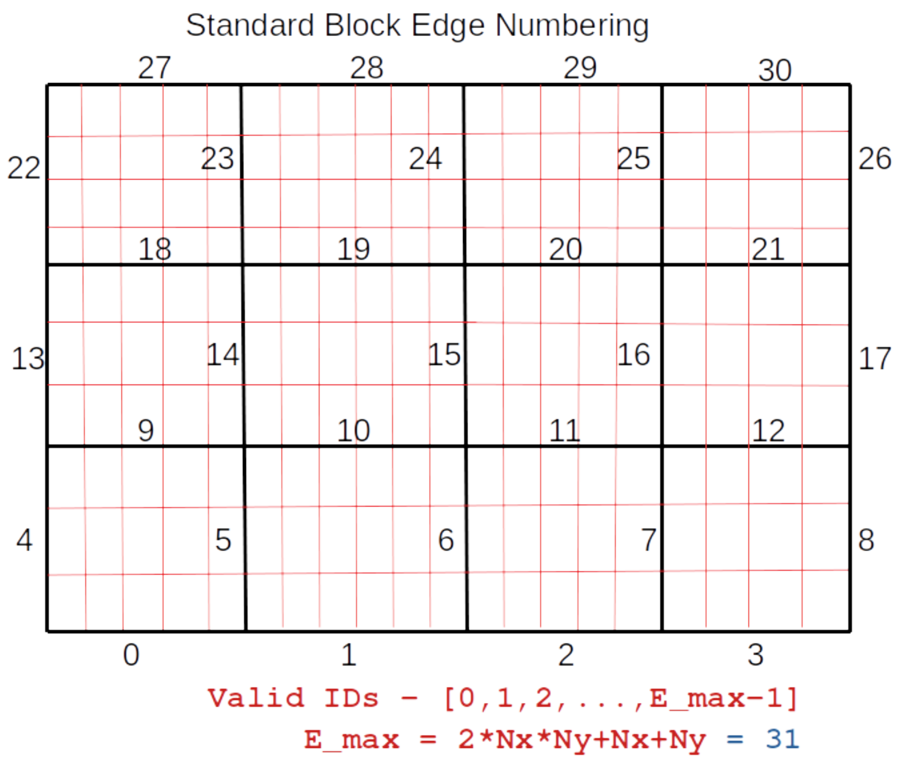
4.3 Block Entity Classification

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

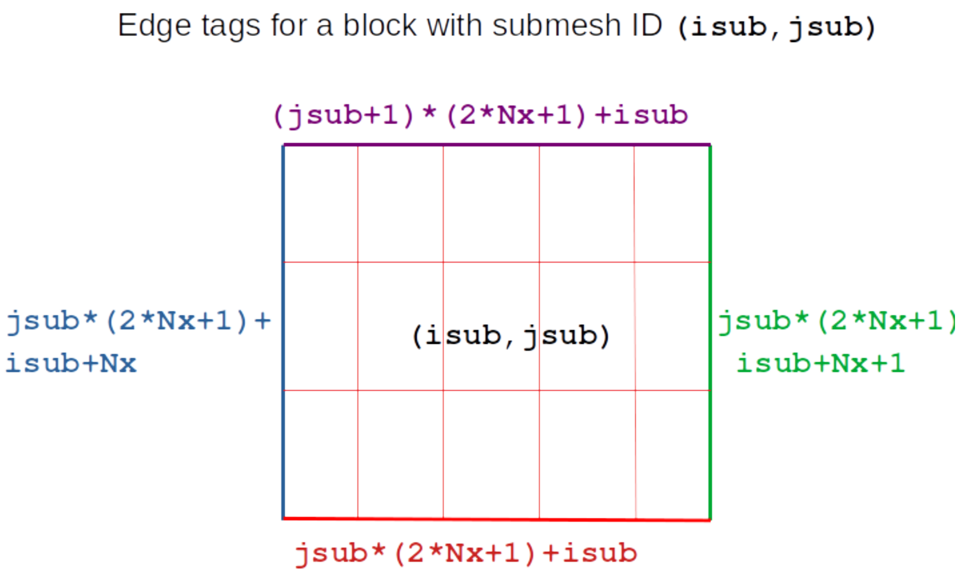


A vertex is a *zero-dimensional entity* while the edge is an *one-dimensional entity.* In order to apply different boundary/interface conditions at different locations, we need to ID these entities. In the following sections the convention used for edge and vertex numbering is explained. The convention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

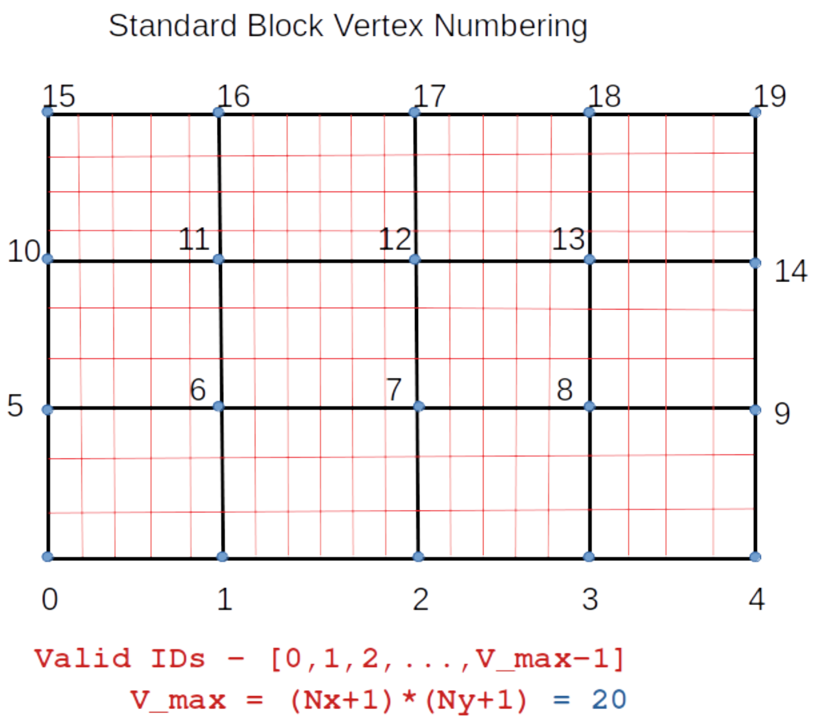
Standard Block-Vertex Numbering


Standard Block Edge Numbering
27
28
29
30
22
23
24
25
26
18
19
20
21
13
14
15
16
17
9
10
11
12
4
5
6
7
8
0
1
2
3
Valid IDs - [0,1,2,...,E_max-1]
E_max = 2*Nx*Ny+Nx+Ny = 31

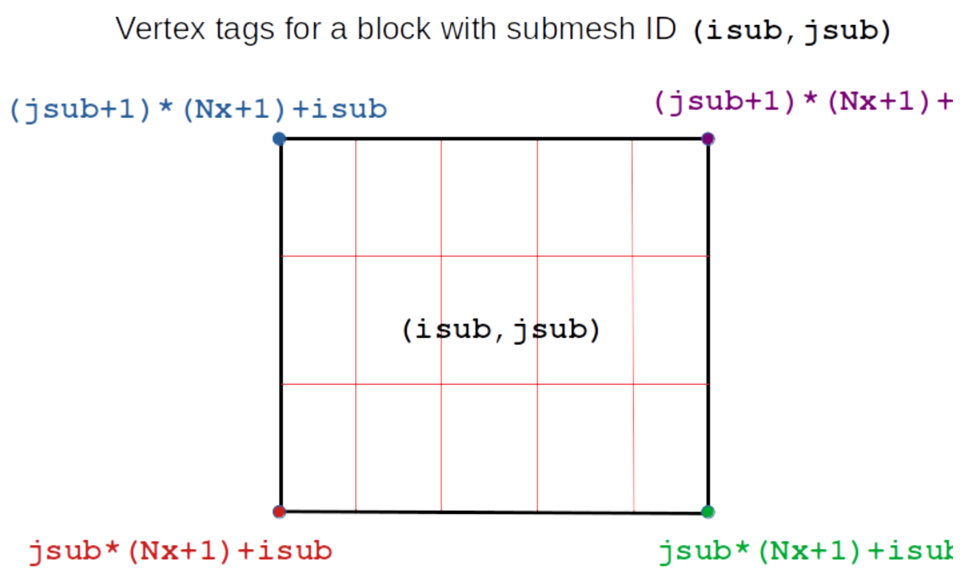
The standardized ID-ing helps us compe up with expressions to obtain the edge IDs of any given block as shown below



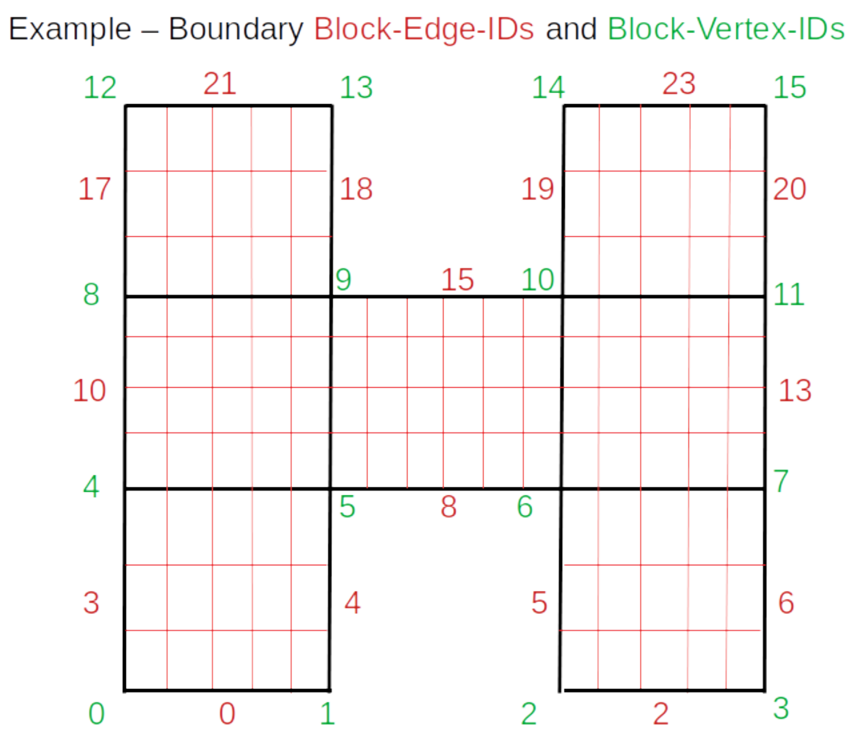
Standard Block-Edge Numbering


Standard Block Vertex Numbering
15
16
17
18
19
10
11
12
13
14
5
6
7
8
9
0
1
2
3
4
Valid IDs - [0,1,2,...,V_max-1]
V_max = (Nx+1)*(Ny+1) = 20

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

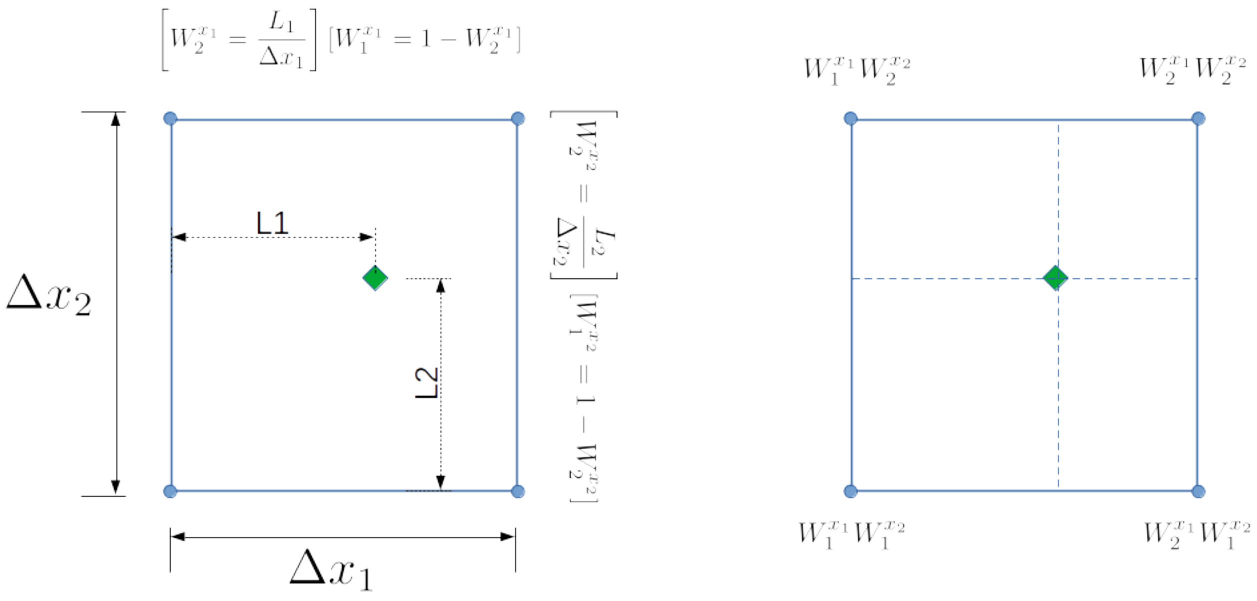


For a mesh with inactive blocks, the scheme remains unchanged i.e. the tags corresponding entities in non-boundary regions will be invalid



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 (see description in next section for more details)

5 2D weight calculations from 1D linear weights



$\left[W_2^{x_1} = \frac{L_1}{\Delta x_1}\right]$ $[W_1^{x_1} = 1 - W_2^{x_1}]$
L1
Δx_2
L2
$\left[W_2^{x_2} = \frac{L_2}{\Delta x_2}\right]$ $[W_1^{x_2} = 1 - W_2^{x_2}]$
Δx_1
$W_1^{x_1}W_2^{x_2}$
$W_2^{x_1}W_2^{x_2}$
$W_1^{x_1}W_1^{x_2}$
$W_2^{x_1}W_1^{x_2}$

Figure 13: (Left) 1D component-wise weight computation
(Right) 1D component-wise weights to 2D weights

6 PUMI-API Usage

6.1 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 thist 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
```

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

6.2 Field-related APIs

- The grading ratio (along a given direction) about an 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 provided as inputs i.e. `knode_x1, knode_x2`, then provide `pumi::elem_on_min_side_offset` for element on min-size 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