ILLINOIS ROCSTAR

SurfMap User’s and Developer's Guide

Illinois Rocstar LLC
1800 S. Oak, STE 208
Champaign, IL 61820

1.0 Introduction

1.1 *SurfMap* Overview

SurfMap provides a small set of commonly used functions for communication among mesh panes in the *COM* infrastructure. It is implemented in C++ and uses the *COM* infrastructure. Typically, *SurfMap*'s static member functions are called from within other *COM* modules. Nevertheless, an interface is provided for using *SurfMap* as a *COM* module. It builds on all platforms supported by *Rocstar* 3.

1.2 Related Documents

The information in this guide is supplemented by the following documents:

- "*COM* User's Guide".
- "*SimIO* User's Guide".

2.0 Purpose and Methods

SurfMap aids in the development of *COM* modules dealing with partitioned meshes by providing commonly required inter-pane communication functions. An API provides access to a few of these functions when the *SurfMap* module is loaded into the *COM* infrastructure. Other functionality is available at a lower level through C++ classes. See 3.3 discusses *SurfMap* as a *COM* module. The important *SurfMap* classes are discussed in this section.

2.1 Dual_connectivity

Element connectivity tables are a fundamental piece of a *COM* mesh. These tables list the nodes which constitute each particular element. The dual connectivity table is a similar structure which lists elements contained in a particular node. *COM* does not require the registration of a dual connectivity table, but this class is available for compiling and accessing that information.

2.1.1 Constructors

```
explicit Pane_dual_connectivity( const COM::Pane *p, bool with_ghost = true)
```

Construct a Pane_dual_connectivity for pane **p**. If **with_ghost** is *true*, then include ghost entities in the construction.

2.1.2 Important Member Functions

`void incident_elements(int node_id, std::vector<int>& elists)`

Fills **elists** with the list of elements containing the node with local id **node_id**.

2.2 Pane_boundary

Determines which nodes are on pane boundaries and which nodes are isolated. An Isolated node is one which does not belong to any element. This can happen when a surface mesh is extracted from a volume mesh. If an element of the volume mesh touches the surface with only a single node, then that node will be isolated in the surface mesh.

2.2.1 Algorithms

Isolated nodes are determined in a straight forward fashion. The isolated node vector is initialized to false for every node. Then, a pass is made over the element connectivity table(s) during which every node seen is marked as not isolated.

Determining which nodes (and facets) are on the border is slightly more complicated. Conceptually, this algorithm is simply facet matching. Facets which are not on the border are sandwiched between two elements. If we consider facets as unordered sets of nodes, then these facets are duplicated on the adjacent elements. Therefore, finding unique facets is equivalent to finding the border.

Again, a pass is made over the element connectivity table(s). An inner loop iterates through every facet of the current element. A four-tuple is created from the ids of the facet's constituent nodes. If the facet is triangular, then *-1* is used as the fourth node id. We compare the four-tuple to the set of all those already seen. If we find a duplicate, then both four-tuples are discarded. After the entire element connectivity is processed, we are left with the set of border faces from which computing border nodes is trivial.

2.2.2 Constructors

`Pane_boundary( const COM::Pane *p)`

Build a Pane_boundary for pane **p.**

`Pane_boundary( const Simple_manifold_2 *pm)`

Build a Pane_boundary for the pane associated with the given manifold.

2.2.3 *Other member functions*

```
determine_border_nodes(
        std::vector<bool> &is_border,
        std::vector<bool> &is_isolated,
        std::vector<Facet_ID > *b=NULL,
        int ghost_level=0) throw(int)
```

Sets **is_border**[node_id -1] to *true* if the node assocaited with the id is on the border, and to *false* otherwise. Stores similar information for isolated nodes in **is_isolated**. If **b** is not NULL, then it is filled with the set of faces which are on the pane border. The interpretation of **ghost_level** is differenrt for structured and unstructured meshes. If the target pane has a structured mesh, it determines how many ghost layers to include in the calculation. If **ghost_level** > 0 and the mesh is unstructured, the all registered ghost entities are considered.

```
static void determine_borders(
        const COM::DataItem *mesh,
        COM::DataItem* is_border,
        int ghost_level = 0)
```

Determines border nodes for the entire window associated with **mesh** and stores this information in **is_border**. The **ghost_level** parameter is the same as in `determine_border_nodes`. Note that the DataItem stored is still *pane* border, and not the border of the mesh as a whole.

2.3 Pane_ghost_connectivity

Adds a single layer of ghost elements to each pane by constructing and registering the complete 5-block pconn for the mesh as defined in the *COM* User's Guide. This is a single layer of ghost elements in the sense that all non local elements incident on a local real node are ghosted as are any nodes which are contained in these elements but not local to the pane. In Rocflu's terminology, this is a second-order mesh.

2.3.1 *Algorithms*

The basic idea of this algorithm is that each pane determines which local elements should be ghosted on other panes, and sends the connectivity information for those local elements to the appropriate panes. Shared node information is built using the Pane_connectivity class, then each local element is examined. If the local element contains a shared node, then the element is added to a set of elements which needs to be sent to the pane(s) with which that node is shared. After all elements have been examined, we can calculate how much information is to be sent to each adjacent pane. This size information is communicated, followed by the actual connectivity data. After all connectivity information is received, the pconn is constructed and registered.

COM itself does not use a global order for nodes or elements, however, we need that information because we will sometimes receive elements from multiple panes which share the same non local node, and we have to recognize that this is a single ghosted node rather than two separate nodes.

2.3.2 Constructors

```
Pane_ghost_connectivity( COM::Window *window)
```

Construct a Pane_ghost_connectivity for window **window.**

2.3.3 Important Member Functions

```
void build_pconn()
```

This is the highest level function, it builds and registers the pconn.

```
void init()
```

The initialization routine builds and registers the shared-node section of the pconn. It also determines border nodes, obtains a list of communicating panes, and creates data structures for the total node ordering.

```
void get_node_total_order()
```

Obtains a total ordering of nodes in the form of an ordered pair <P,N> where P is the “owner pane” and N is the node’s local id on the owner pane. The “owner pane” is the pane with highest id which contains a real copy of the node. For each node, N is initialized to 0 and P is initialized to the local pane id. Then, the shared node section of the pconn is examined. P is updated each time a node is found being shared with a pane with a higher pane id than the currently stored value. At this point, all the owner panes have been determined. Each node owned by the local pane is now assigned its local node id as the N value. An MPI maximum reduction is performed on the N values to finalize the total ordering.

```
void get_ents_to_send(
     vector<vector<vector<int> > > &gelem_lists,
     vector<vector<map<pair<int,int>,int> > > &nodes_to_send,
     vector<vector<deque<int> > > &elems_to_send,
     vector<vector<int> > &comm_size)
```

This function determines the local elements and nodes to be remotely ghosted on communicating panes. A dual connectivity data structure is built which allows for querying which elements are adjacent to each node. The list of nodes shared with each communicating pane is examined, and all elements containing any of these nodes are added to a set to be sent to the communicating pane. Simultaneously, the eventual size of the element connectivity information to send is calculated and stored. Each elements type and node list (as <P,N> pairs) will be sent. The actual information to be send is written to **gelem_lists**, while the lists of the local nodes and elements to be remotely ghosted are stored in **nodes_to_send** and **elems_to_send**.

COM requires that the elements listed in one pane’s real-elements-to-send list correspond to the element listed in the same index of the communicating pane’s ghost-elements-to-receive list, and similarly for the ghost node communication sections. We fulfill this requirement as follows.

Owner panes list real-elements-to-receive in the same order that the elements are placed in the **gelem_lists** data structure, and the pane which ghosts the elements maintains this ordering. Similarly, as element connectivities are written into the **gelem_lists** structure, the owner pane checks to see if the node is already shared with the communicating pane. If not, then that node will need to be ghosted on the remote pane. These nodes are listed in the real-node-to-share list in the same order that they are first observed. When the remote pane processes the element connectivity lists, the same method is used.

```
void process_received_data(
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &elem_renumbering,
      vector<vector<map<pair<int,int>,int> > > &nodes_to_recv)
```

Determines the number of ghost nodes to receive, assigns them local ghost node ids, and maps <P,N> to those ids. Also determine the number of ghost elements of each type to receive. Ghost element ids can not be determine as elements arrive if we want to have a single connectivity table per element type because *COM* requires that elements in a single connectivity table be numbered consecutively.

```
void finalize_pconn(
      vector<vector<map<pair<int,int>,int> > > &nodes_to_send,
      vector<vector<map<pair<int,int>,int> > > &nodes_to_recv,
      vector<vector<deque<int> > > &elems_to_send,
      vector<vector<int> > &elem_renumbering,
      vector<vector<vector<int> > > &recv_info)
```

Takes the data we've collected, and builds and registers the pconn. The shared node information is already present, so there are four blocks to build: real-nodes-to-send, ghost-nodes-to-receive, real-elements-to-send and ghost-elements-to-receive. Data for the first three of these blocks is available in the correct order from the first three input parameters to the function. Creating the ghost-elements-to-receive block of the pconn requires combining information in the latter two data structures.

First, the input data structures are examined to determine the size of each ghost block of the pconn. Connectivity tables are resized to accommodate the new ghost entries, and the pconn is resized to accommodate the four new blocks. Next we actually fill in the new sections of the pconn. We also fill in the ghost element's connectivity into the correct connectivity table by taking into account both the element's type and the order in which it was received. Finally, we extend the nodal coordinate DataItem to accommodate the new ghost nodes, and update those coordinates from their real coordinates using the newly created pconn.

```
void get_cpanes()
```

Get the list of communicating panes for each local pane. A communicating pane is any pane with which this pane shares a node. This data is stored in **_cpanes**.

```
void send_gelem_lists(
      vector<vector<vector<int> > > &gelem_lists,
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &comm_sizes)
```

Send ghost element connectivity lists to the communicating panes.

```
void send_pane_info(
      vector<vector<vector<int> > > &send_info,
      vector<vector<vector<int> > > &recv_info,
      vector<vector<int> > &comm_sizes)
```

Sends an arbitrary amount of data to the communicating panes.

```
void determine_shared_border()
```

Determines whether or not each local node is shared.

```
void mark_elems_from_nodes(
       vector<vector<bool> > &marked_nodes,
       vector<vector<bool> > &marked_elems)
```

Takes the given Boolean nodal property, and extends that property to elements. An element is considered to have the property if any of its constituent nodes have the property.

3.0 Building and Running

SurfMap is written in C++ and uses the *CMake* infrastructure with *MPACT.* The source and CMakeLists.txt are found at /MPACT/SurfMap in the *MPACT* distribution. The *SurfMap* module is built automatically by the *MPACT* build system as libSurfMap.so

The former contains utility programs automatically built by the makefiles, while the latter contains all libraries including *COM*'s and *SurfMap*'s.

3.1 Library Dependencies and Building

SurfMap is integrated into *MPACT* and is built along with it. Because *SurfMap* links to the *COM* library, it may only be built directly from its own directory if the *COM* library already exists.

3.2 *SurfMap* Build Targets

The default *COM* build creates the *SurfMap* dynamic library as well as a *SurfMap* utility named “addpconn”. Several other test programs are included in *SurfMap*/test, and may be built individually:

Test Program	*Description*
bordertest_hex	Demonstrates determination of pane borders through *SurfMap* on an unstructured hex mesh. Builds an unstructured hex mesh and uses *SurfMap* to determine which nodes are on the border. Dumps the mesh into an .hdf file, “hexmesh”, with node border information stored in the DataItem “borders”.
bordertest_struc	Demonstrates determination of pane borders through *SurfMap* on a structured hex mesh. Creates a structured hex mesh, determines which nodes are on the pane border, and dumps the mesh into an .hdf file, “strucmesh”

3.3 *COM* Accessible Functions (*SurfMap* API)

This section describes the set of functions which is available through the *COM* infrastructure when *SurfMap* is registered as a module. All of these functions are *static void* member function of the *SurfMap* class.

compute_pconn(
const COM::DataItem *mesh,
COM::DataItem *pconn)

Computes the first block of the pconn DataItem described in the *COM* User's Guide. If pconn hasn't been initialized, then memory is allocated and the computed pconn block is saved. Otherwise, saves up to the capacity of the DataItem.

Parameter	*Description*
*mesh	The target mesh.
*pconn	The target mesh's pconn DataItem.

pane_border_nodes(
const COM::DataItem *mesh,
COM::DataItem *isborder,
int *ghost_level=NULL)

Determines which nodes are on the pane border, and saves this information to a *COM* DataItem.

Parameter	*Description*
*mesh	The target mesh.

Parameter	***Description***
*isborder	DataItem where border information will be saved.
*ghost_level	If > 0, include ghost nodes and elements as part of the pane.

reduce_average_on_shared_nodes(
COM::DataItem *att,
COM::DataItem *pconn = NULL)

Calculates an average DataItem value for each shared node across all sharing panes, and sets the DataItem value to that average on all sharing panes.

Parameter	***Description***
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to the target DataItem

reduce_maxabs_on_shared_nodes(
COM::DataItem *att,
COM::DataItem *pconn=NULL);

Sets the value of each component of an DataItem to the value of that component of the DataItem with the largest magnitude. Note that this is done on a *component by component basis*, not in the sense of any norms.

Parameter	***Description***
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to the target DataItem

update_ghosts(
COM::DataItem *att,
COM::DataItem *pconn = NULL)

Sets the value of an DataItem at ghost nodes or cells to the value on the corresponding real nodes or cells. In the case that a shared node has different values across its incident panes, it is undetermined which value each ghost node will receive.

Parameter	***Description***
*att	Target DataItem
*pconn	Pconn of the mesh corresponding to target DataItem

3.4 Other Functions (*SurfMap* API)

SurfMap()

SurfMap's constructor, does not perform any initialization.

static void load(const std::string &mname)
static void unload(const std::string &mname)

These functions are used for loading or unloading *SurfMap* from *COM* with the given module name.

4.0 Input and Output (User Interface)

SurfMap if typically used as a C++ object, though its functions other than the constructor are all static, so no instantiation is required. It may also be loaded as a *COM* module, and called through the standard *COM* interface.

4.1 *SurfMap* as a C++ Object

The following code fragment demonstrates the use of *SurfMap* as a C++ object:

```
// Assuming that "window" is a pointer to a COM window
// move shared nodes to their average position across all
// incident panes
SurfMap::reduce_average_on_shared_nodes(window->
     DataItem(COM::COM_NC));

// Update the ghost copies of the nodes with their new positions
SurfMap::update_ghosts(window->DataItem(COM::COM_NC),
     window->DataItem(COM::COM_PCONN));
```

4.2 *SurfMap* as a *COM* Module

SurfMap is typically loaded and invoked through the *COM* API as illustrated in the following C++ example:

```
// Load SurfMap into the COM infrastructure
COM_LOAD_MODULE_STATIC_DYNAMIC( SurfMap, "MAP");

// Get function handle for SurfMap::compute_pconn
int MAP_compute_pconn =
     COM_get_function_handle("MAP.compute_pconn");
```

```
// Get the handle for the fluids mesh
int mesh_hdl = COM_get_DataItem_handle("fluids.mesh");

// Get the handle for the pconn DataItem of the fluids mesh
int pconn_hdl = COM_get_DataItem_handle("fluids.pconn");

// Use SurfMap::compute_pconn to create the shared-node section
// of the pconn for the fluids mesh.
COM_call_function(MAP_compute_pconn, &mesh_hdl, &pconn_hdl);
```

5.0 Utilities and Test Programs

A default build of *SurfMap* creates "addpconn", a utility for building the shared-node section of pconn for a given .hdf file. Source code for this utility is found in *SurfMap*/util/addpconn.C . Four other test programs are available in *SurfMap*/test/ . The source files for these programs also have the same name as the programs, but with ".C" appended. Section 3.2 explains how to build them.

5.1 addpconn

This utility reads in one or more .hdf files using *SimIN*. It makes a copy of the mesh from which it removes all mesh DataItems other than nodal coordinates and connectivity tables. *SurfMap* is then used to rebuild the pconn on the new mesh, which is written to file.

```
addpconn <input filename patters or Rocin control file>
         <output file prefix>
```

To run in parallel, a *SimIN* control file must be passed as a second argument. If the second argument ends in ".hdf", then all panes are written out to a single pane. Otherwise, each pane is written to a separate .hdf file.

5.2 bordertest_hex

This test program runs without any command line input. It builds an unstructured hex mesh with 4 elements and 18 nodes. *SurfMap* is used to determine which nodes are on the border, and store this information in an DataItem. The mesh is then written out to "hexmesh0000.hdf".

5.3 bordertest_struc

This program is similar to bordertest_hex, except that a much larger mesh is created, and it is a structured hex mesh. The output file is "strucmesh0000.hdf".