

STK-Mesh Example Dynamic Mesh Modification

**Carter Edwards, Todd Coffey,
Dan Sunderland, Alan Williams**





Outline

- **The Meta-Data**

- **Modifying the Bulk-Data**
 - Maintaining Consistency
 - ◆ Bulk-Data State
 - ◆ Parallel Consistency
 - ◆ Local Consistency
 - ◆ Field Data
 - ◆ Atomic Modifications

- **Gear Demo**
 - Overview
 - Code snippets necessary to strip teeth off of the gear





Mesh Modification

■ Meta-Data

- Equivalent to the schema of a database, the meta-data describes the problem domain
- Freely modifiable before being committed
- **No** modifications allowed after commit





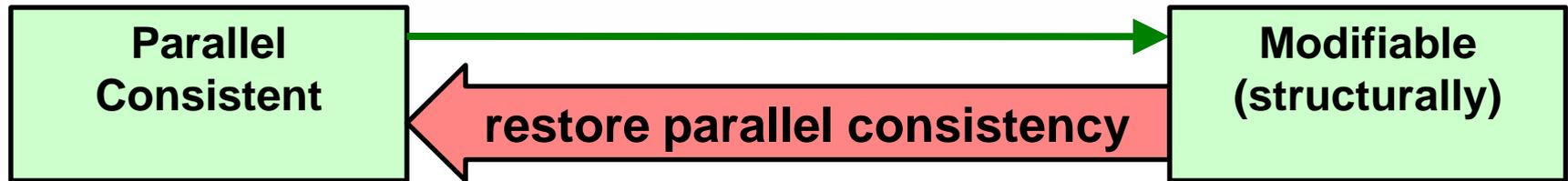
Mesh Modification

■ Bulk-Data

- Contains the discretization of the problem
- Its meta-data ***must be*** committed before any function beside the bulk-data constructor can be called
 - ◆ *Meta-data check for global consistency will throw if the meta-data is not globally consistent*
- Bulk-data modifications are only allowed when the bulk-data is in a modifiable state.



Modification Cycle



- **Parallel Consistent**
 - **NO** Mesh modification allowed
- **Modifiable**
 - Guaranteed to be locally consistent
 - Atomic mesh modifications are allowed

*Parallel consistency is enforced when switching from **Modifiable** to **Parallel Consistent***

Maintaining Consistency

- The bulk-data state is transitioned by calling

```
bulk_data.modification_begin()  
bulk_data.modification_end()
```

- A new modification cycle begins whenever modification begin is called
- Atomic mesh modifications mark affected entities and all their upward relations as *modified*
- Parallel consistency is enforced at modification end by
 - ◆ Deleting all ghosts of modified entities
 - ◆ Resolving parallel ownership and sharing of created and destroyed entities
 - ◆ Resolving shared entities mesh part membership, entity relations, field data memory allocation, and bucket membership
 - ◆ Updating the one layer ghosted aura





Maintaining Consistency

- **The following operations are available through Atomic Mesh Modifications**
 - Creating and/or deleting entities
 - Changing entities' mesh part memberships
 - Changing entities' relations
 - Moving entities' ownership to another process

- **Atomic modifications are guaranteed to be *locally consistent***
 - Induced mesh part membership will change as necessary
 - Memory for field data will be created, resized, or deleted
 - Existing field data will move to the correct bucket



Maintaining Consistency

Field Data

- Communicating field data **does not** modify the topology of the mesh and can happen at any time (i.e. not restricted to a modifiable bulk-data)
- Atomic mesh modifications will move existing field data to the correct bucket on the *local* process
 - When **changing entity owner** the field data is moved to the correct bucket on the remote process
- After modification end space for field data of ghosted and shared entities has been allocated but the data *has not been copied* from the owner
- Field data values can be copied from owned to shared/ghosted at any time by calling

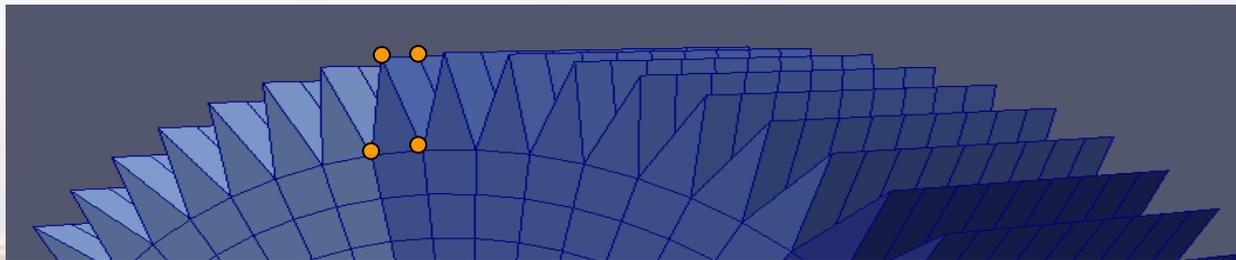
```
bulk_data.communicate_field_data(...);
```



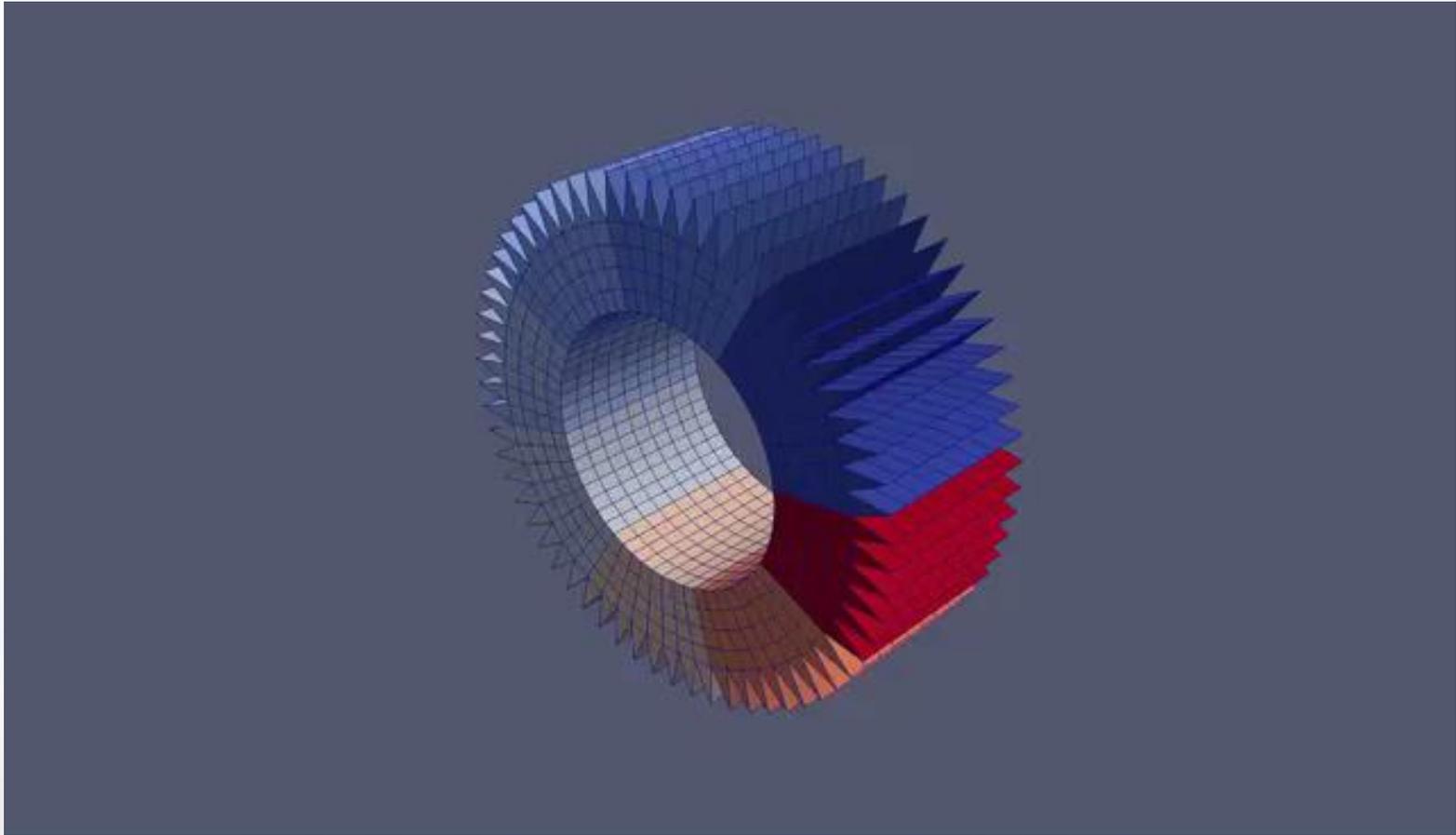
Simple Mesh Modification Example

■ Breaking teeth off of the gear

- Distribute mesh over available processes
- Create 6 new nodes to attach to the wedge
- Destroy the relationship between the current nodes and the wedge
- Attach the new nodes to the wedge
- Copy the field data from the current nodes to the new nodes
 - ◆ The current nodes stay with the body gear



Putting it all together



Distribute mesh across available processes

- **Change entity owner (moving the entity to another process)**
 - The bulk data function `change_entity_owner` is a parallel collective call that ***gives away*** ownership of entities to other processes

```
//EntityProc is the pair
//(Entity *owned_entity, unsigned to_proc_id)
std::vector< stk::mesh::EntityProc> > move_entities;

// move the wedge to proc 2
if ( bulk_data.parallel_rank() == 0) {
    move_entites.push_back( std::pair(&wedge,2));
}

// Parallel collective call
bulk_data.change_entity_owner( move_entities );

// The field data will also be moved with the wedge
```



Creating a new wedge

■ Creating new entities

- When new entities are created the creating process is the *owner* of the entity
- If two or more processes create an entity with the same rank and identifier, the entity will be ***shared***

```
//Declare a new wedge element
stk::mesh::PartVector add_parts;
add_parts.push_back( wedge_part );
stk::mesh::Entity & wedge = bulk_data.declare_entity(
                                                    element_rank,
                                                    element_id,
                                                    add_parts
                                                    );
```



Creating new nodes

■ Generating new entities

- The bulk data function `generate_new_entities` is a parallel collective call that will create new entities of the requested ranks with *globally unique ids*

```
//create 6 nodes on process 0
std::vector<size_t> num_requested_entities(num_entities_rank,0);

if ( bulk_data.parallel_rank() == 0) {
    num_requested_entities[node_rank]=6;
}

stk::mesh::EntityVector requested_nodes;

//parallel collective call
bulk_data.generate_new_entities (
                                num_requested_entities,
                                request_nodes
                                );
```



Attaching the nodes to the wedge

■ Creating/Destroying entity relations

- Relations are directed from the higher ranking entity to the lower ranking entity, the converse relation is automatically inserted/deleted
- Processes can only create/destroy relations when they own or share the higher ranking entity
- Creating or deleting relations may change entity's induced mesh part membership

```
// declare relations from the wedge to the nodes
for ( size_t i=0; i<6; ++i) {
    bulk_data.declare_relation (
        wedge, // from entity
        requested_nodes[i], // to entity
        i // relation identifier
    );
}
```



Destroying nodes

■ Destroying entities

- A process may destroy its reference to an entity if the entity does not have a relation to a higher ranking entity in its owned closure.
- If the owning process destroys an entity and another process *shares* the entity, ownership automatically transfers to a *sharing* process

```
// destroy the old nodes attached to the wedge
stk::mesh::PairIterRelations
    node_relations = wedge.relations(node_rank);

for ( size_t i=0; i<node_relations.size(); ++i) {
    stk::mesh::Entity * node = node_relations[i].entity();
    //destroy relation to wedge
    bulk_data.destroy_relation( wedge, *node );
    //destroy the entity
    bulk_data.destroy_entity( node );
}
```



Changing mesh part membership

- Processes can only change part membership when they **own** or **share** the entity
- The entity will be moved to a different bucket
- The fields available to the entity will change to match the fields restrictions

```
// add the wedge to the cylindrical coordinate part
stk::mesh::PartVector add_parts, remove_parts;
add_parts.push_back( & cylindrical_coord_part );

bulk_data.change_entity_parts(
    wedge,
    add_parts,
    remove_parts
);

// If cylindrical_coord_part was declare to be of rank element,
then the nodes will be induced into this part and the
cylindrical coordinate field will be available to the nodes
```



Gear Demo: Putting it all together

