SAND2014-17258R

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Converting Projects from STK Classic to STK

James Foucar

19 August 2014

Intro

The version of STK (Sierra ToolKit) that has long been provided with Trilinos is no longer supported by the core development team. With the introduction of a the new STK library into Trilinos, the old STK has been renamed to stk_classic. While stk_classic is still available and functional, all users of stk_classic are strongly encouraged to port their code to STK for the following reasons:

- stk_classic is no longer support by the team that created it (no bugfixes, no user support)
- The new STK offers improved performance and smaller memory footprint for most usecases
- The new STK offers an improved API

This document contains a rough guide of how to port a stk_classic code to STK.

Trivial

Easy changes that can be made via sed or brain-dead human churn:

- Changes to includes:
  - Include `stk_mesh/base/FieldBase.hpp` instead of `stk_mesh/base/FieldData.hpp`
    * `sed -r -i 's/FieldData.hpp/FieldBase.hpp/g' *`
  - Any `stk_mesh/fem/*.hpp` include should be changed to `stk_mesh/base/*.hpp`
    * `sed -r -i 's:stk_mesh/fem/:stk_mesh/base/:g' *`
- The fem namespace doesn't exist anymore, just use `stk::mesh`
  - `sed -r -i 's/::fem//g' *`
- If you had already switched to stk_classic, switch the namespace back
  - `sed -r -i 's/stk_classic/stk/g' *`
- Any comparison between an Entity and NULL should be replaced with `bulk_data.is_valid(entity)`

Non-Trivial

Entities as a mesh index

One of the most important conceptual changes from stk_classic to stk is the transformation of the Entity from a first-class object, holding and managing it's own data, to a simple mesh index, basically no more than an integer. With this change, it is strongly recommended that clients hold and pass Entities by value (not by reference or pointer) in order to avoid unnecessary pointer hops.

With this change, almost all member functions have been removed from Entity. Instead, accessing Entity data must be done through BulkData. A very common pattern is the following:

```
// Getting the ID of an Entity:

//   STK CLASSIC:
Entity const& entity = ...;
int id = entity.identifier();

//   NEW STK:
BulkData & bulk_data = ...;
Entity entity = ...;
int id = bulk_data.identifier(entity);
```

While I have yet to come up with an automated system that's smart enough to do the conversion above, I do have a sed expression can help transform all Entity pointers and reference types to value types:

```
sed -r -i 's/(const +)?stk::mesh::Entity ?(const *)?[*&]/stk::mesh::Entity/g' *
```

Note, the above sed expression will not work for code that was using Entity without the full namespace included.

Connectivity/Relations

The second-most important conceptual change from stk_classic to stk deals with Entity relations. First, in the API, what used to be called a *relation* is now called *connectivity*. The management of connectivity is done in a completely different data structure. In stk_classic, connectivity was managed in a very object-oriented fashion, with each Entity containing its own vector of its connectivity. With the transformation of Entities to mesh indices, the Entity connectivity data is stored in large blocks within Buckets. With these changes, the API for getting Entity connectivity is very different:

```
// Iterating nodal connectivity of an element:

//   STK CLASSIC:
Entity const& element = ...;
const PairIterRelation &relations = element.relations(NODE_RANK);
for (size_t i = 0; i < relations.size(); ++i) {
  Entity const& node = relations[i].entity();
  // do something with node
}

//   NEW STK:
BulkData & bulk_data = ...;
Entity element = ...;
Entity const* relations = bulk_data.begin_nodes(element);
size_t const num_nodes = bulk_data.num_nodes(element);
for (size_t i = 0; i < num_nodes; ++i) {
  Entity node = relations[i];
  // do something with node
}
```

Things to note here:

- Connectivity is retrieved from the BulkData, not the Entity
- PairIterRelation is gone. Instead, raw pointers to the data are provided.
- Since only a raw pointer is provided, an additional call to get the length of the data is needed.
- The Relation class, which encapsulated all data for a relation, is gone.

For the vast majority of cases, clients are only interested in the Entity targets of the connectivity, but in some cases the other pieces of data, ordinals and permutations, are of interest:

```
// Iterating edge connectivity of an element, including ordinal and permutation data:

//   STK CLASSIC:
Entity const& element = ...;
const PairIterRelation &relations = element.relations(EDGE_RANK);
for (size_t i = 0; i < relations.size(); ++i) {
```

```
  Entity const& edge = relations[i].entity();
  RelationIdentifier identifier rel_id = relations[i].identifier();
  unsigned rel_permutation = relations[i].permutation();
  // do something
}

//   NEW STK:
BulkData & bulk_data = ...;
Entity element = ...;
Entity const* relations = bulk_data.begin_edges(element);
ConnectivityOrdinal const* ordinals = bulk_data.begin_edge_ordinals(element);
Permutation const* permutations = bulk_data.begin_edge_permutations(element);
size_t const num_edges = bulk_data.num_edges(element);
for (size_t i = 0; i < num_edges; ++i) {
  Entity edge = relations[i];
  ConnectivityOrdinals ord = ordinals[i];
  Permutation perm = permutations[i];
  // do something
}
```

As we can see having to make a call to BulkData for each type of connectivity data makes this use case slightly less convenient than before.

The last important thing to note is that additional topological invariants are enforced in the new API:

- The number of connectivity cannot exceed topological limit
  - E.g. a hex8 element cannot have more than 8 nodes
- The ordinals of downward (higher rank to lower rank) connectivity must fall in a logical range
  - E.g. a hex8 element connectivity to a node must have ordinal 0-7
- Nodal connectivity must be complete
  - E.g. a hex8 element must have exactly 8 nodes
- Permutation data is only allocated when it makes sense
  - E.g. nodal connectivity does not have permutation data

Field changes

The Field API has not changed too much. The main change was that field *declarations* now require an entity rank and field *restrictions* do not. What this implies is that a field may only be associated with a single entity rank. If you had a field that was being put on multiple types of entities, you'll have to split that field into multiple fields (one per rank).

```
// Declare a field of double and putting it on all nodes

typedef Field<double> FieldType;
MetaData & meta_data = ...;
std::string field_name = ...;

//   STK CLASSIC:
FieldType& field = meta_data.declare_field<FieldType>(field_name);
put_field(field, meta_data.node_rank() , meta_data.universal_part());

//   NEW STK:
FieldType& field = meta_data.declare_field<FieldType>(stk::topology::NODE_RANK, field_name);
put_field(field, meta_data.universal_part());
```

BucketArray removal

Another important thing to note about Fields is that the BucketArray format for retreiving field data is no longer available. The idea behing BucketArrays was to automatically convert data for a multidimensional Field into a shards multidimensional array. Users will have to manage their own striding over multidimensional data.

```
// Get data for a 2-dimensional field and iterate over it

typedef Field<double, Tag1, Tag2> FieldType;
FieldType& field = ...;
Bucket& bucket = ...;

//   STK CLASSIC:
typedef BucketArray<FieldType> BAType;
size_t const num_field_dims = field.rank();
BAType bucket_array( field, bucket );
for (int i=0; i < bucket_array.dimension(0); ++i) {
  for (int j=0; j < bucket_array.dimension(1); ++j) {
    for (int k=0; k < bucket_array.dimension(2); ++k) {
      double data = bucket_array(i, j, k);
    }
  }
}

//   NEW STK:
size_t const num_field_dims = field.field_array_rank();
double const* data = field_data( field, bucket );
size_t const num_per_entity = field_scalars_per_entity(field, bucket);
size_t const dim1 = Tag1::Size;
size_t const dim2 = Tag2::Size;
for ( size_t k= 0; k < bucket.size(); ++k) {
  double const* data_for_entity = data + (k * num_per_entity);
  for (int i=0; i < dim1; ++i) {
    for (int j=0; j < dim2; ++j) {
      double data = data_for_entity[dim2*i + j];
    }
  }
  // do something
}
```

I/O

If you were using the I/O convenience wrapper MeshReadWriteUtils, you'll need to port all code that was using that class to use StkMeshIoBroker. The interfaces and behaviors of the two classes are similar but not identical.

```
// Reading a mesh

std::string file_name = ...;
MPI_Comm comm = ...;
std::vector<std::string> entity_rank_names = ...;

//   STK CLASSIC:
MetaData& meta_data = ...;
stk::io::MeshData mesh_data;
stk::io::create_input_mesh("exodusII",
                           file_name,
                           comm,
                           meta_data,
                           mesh_data,
                           entity_rank_names);
```

```
BulkData& bulk_data = ...;
stk::io::populate_bulk_data(bulk_data, mesh_data);
int index = ...;
stk::io::process_input_request(mesh_data, bulk_data, index);

//   NEW STK:
stk::io::StkMeshIoBroker mesh_data(comm);
mesh_data.set_rank_name_vector(entity_rank_names);
size_t mesh_index = mesh_data.add_mesh_database(file_name, "exodusII", stk::io::READ_MESH);
mesh_data.set_active_mesh(mesh_index);
mesh_data.create_input_mesh();
MetaData& meta_data = mesh_data->meta_data();
// * declare input fields on meta_data * //
BulkData& bulk_data = ...;
mesh_data.set_bulk_data(bulk_data);
mesh_data.add_all_mesh_fields_as_input_fields();
mesh_data.populate_bulk_data();
int index = ...;
mesh_data.read_defined_input_fields(index);
```

Important items to note in the above code:

- StkMeshBroker can handle multiple input databases
- StkMeshBroker generates it's own MetaData which the client must use
- Clients of StkMeshIoBroker must declare input fields on this MetaData
- Clients of StkMeshIoBroker must explicitly add fields as input fields

```
// Writing a mesh

std::string file_name = ...;
MPI_Comm comm = ...;

//   STK CLASSIC:
MetaData& meta_data = ...;
BulkData& bulk_data = ...;
stk::io::MeshData mesh_data;
stk::io::create_output_mesh(file_name,
                            comm,
                            bulk_data,
                            mesh_data);
stk::io::define_output_fields(mesh_data, meta_data);
double time_label = ...;
int out_step = stk::io::process_output_request(mesh_data, bulk_data, time_label);


//   NEW STK:
MetaData& meta_data = ...;
BulkData& bulk_data = ...;
stk::io::StkMeshIoBroker mesh_data(comm);
mesh_data.set_bulk_data(bulk_data);
int mesh_index = mesh_data.create_output_mesh(file_name, stk::io::WRITE_RESULTS);
const stk::mesh::FieldVector &fields = meta_data().get_fields();
for (size_t i=0; i < fields.size(); i++) {
  if (* is fields[i] IO field? *) {
    mesh_data.add_field(mesh_index, *fields[i]);
  }
}
double time_label = ...;
int out_step = mesh_data.process_output_request(mesh_index, time_label);
```

The changes in how a mesh should be output are similar to the input changes. The major takeaways are that StkMeshIoBroker can handle multiple inputs and outputs and that defining input/output Fields must be done manually.

stk::topology replaces shards

While STK still supports the shards topology library, stk::topology is the prefered topology library and support for shards may be discontinued at some point in the future.

The differences between the libraries are too numberous to describe here and a detailed conversion documentation is beyond the scope of this document. If you need help with this, please email me (jgfouca@sandia.gov).

EntityRank changes

EntityRanks are now derived from stk::topology, not MetaData, with one exception: side_rank (depends on spatial dimension). The datatype has also changed from a plain int to an enum which prevent implicit casting of EntityRanks to integer types. In places where an int was being used as an EntityRank, you should use an EntityRank instead. If you must use an int, you'll have to static_cast it to an EntityRank.

```
// Getting various EntityRanks

//   STK CLASSIC:
FEMMetaData & meta_data = ...;
EntityRank node_rank1 = meta_data.node_rank();
EntityRank node_rank2 = FEMMetaData::NODE_RANK; // Alternate
EntityRank edge_rank1 = meta_data.edge_rank();
EntityRank edge_rank2 = FEMMetaData::EDGE_RANK; // Alternate
EntityRank face_rank1 = meta_data.face_rank();
EntityRank face_rank2 = FEMMetaData::FACE_RANK; // Alternate
EntityRank vol_rank1  = meta_data.volume_rank();
EntityRank vol_rank2  = FEMMetaData::VOLUME_RANK; // Alternate
EntityRank side_rank  = meta_data.side_rank();
EntityRank elem_rank  = meta_data.element_rank();

//   NEW STK:
MetaData & meta_data = ...;
EntityRank node_rank = stk::topology::NODE_RANK;
EntityRank edge_rank = stk::topology::EDGE_RANK;
EntityRank face_rank = stk::topology::FACE_RANK;
EntityRank elem_rank = stk::topology::ELEMENT_RANK;
EntityRank side_rank = meta_data.side_rank();
```

Things to note:

- There were two ways to get EntityRanks in the old API, now there's only one
- FEMMetaData is gone, all remaining FEMMetaData methods are now in MetaData
- Element rank was not fixed in the old API; it is fixed in the new API

Access to BulkData

There is now virtually nothing you can do with your mesh without a handle to your BulkData. You can no longer get to your BulkData from an Entity, so there are cases where classes/functions will need to be passed a BulkData when they previously did not need one.

```
// Dumb function to illustrate need for BulkData

//   STK CLASSIC:
int get_entity_id_wrap(Entity const& entity)
{
```

```
  return entity.identifier();
}

//   NEW STK:
int get_entity_id_wrap(Entity entity, BulkData& mesh)
{
  return mesh.identifier(entity);
}
```

Gotchas

The section contains extra information on conversion items that are error-prone in subtle ways.

BulkData constructor

The arguments and default arguments for BulkData have changed but, in certain cases, old BulkData constructions will be compatible with the new signature as far as the compiler is concerned. Semantically, the arguments are not compatible but without the compiler complaining, it's an easy bug to have slip in.

```
// BulkData constructor signatures

//   STK CLASSIC:
BulkData( MetaData & mesh_meta_data ,
          ParallelMachine parallel ,
          unsigned bucket_max_size = 1000 ,
          bool use_memory_pool = true );

//   NEW STK:
BulkData( MetaData & mesh_meta_data
        , ParallelMachine parallel
        , bool add_fmwk_data = false
        , ConnectivityMap const* arg_connectivity_map = NULL
        , FieldDataManager *field_dataManager = NULL
        );
```

Looking at the constructors above, note that both the third and fourth arguments have the potential to be implicitly converted from the old type to the new type. In general, it will be good to review all the places where you're constructing BulkDatas to make sure they're correct.

EntityRank arithmetic

There are important invariants regarding EntityRank values in stk_classic that are no longer true. The most dangerous change is that an Entity's side rank is no longer guaranteed to the rank of the Entity minus one. This is particularly dangerous for 2D code or code that's intended to work in both 2D and 3D.

To phrase the issue differently, in stk_classic, Entity ranks were guaranteed to be sequential. In 2D, node_rank=0, edge_rank=1 (side_rank also is 1), and element_rank=2. In the new stk, this invariant no longer holds. In 2D, node_rank=0, edge_rank=1 (side_rank also is 1), and element_rank=3.

This change makes any code that was doing EntityRank arithmetic fragile. For example:

```
// Get connectivity to immediately lower-ranking entities.

//   STK CLASSIC:
Entity const& entity = ...;
EntityRank rank_below_mine = entity.entity_rank() - 1;
const PairIterRelation &relations = element.relations(rank_below_mine);

//   NEW STK BAD CONVERSION:
```

```
BulkData & bulk_data = ...;
Entity entity = ...;
EntityRank rank_below_mine = static_cast<int>(bulk_data.entity_rank(entity)) - 1;
Entity const* relations = bulk_data.begin(entity, rank_below_mine);

//   NEW STK GOOD CONVERSION:
EntityRank get_rank_below_mine(MetaData& meta_data, EntityRank rank)
{
  if (rank == stk::topology::ELEMENT_RANK)
    return meta_data.side_rank();
  else
    return --rank;
}
BulkData & bulk_data = ...;
MetaData & meta_data = ...;
Entity entity = ...;
EntityRank rank_below_mine = get_rank_below_mine(meta_data, bulk_data.entity_rank(eneity);
Entity const* relations = bulk_data.begin(entity, rank_below_mine);
```

Note that the code in the BAD CONVERSION section will not work in 2D if entity is an element because, in 2D, an element will never have connectivity to entities of rank 2 (FACE_RANK).

I/O

There were a couple issues with I/O that really tripped-up the Albany conversion from stk_classic.

The first issue, and by far the most difficult bug during the Albany conversion, is that old nemesis files will no longer work. If you have access to an up-to-date seacas, you can regenerate nemesis files with this command:

```
% decomp p $NUM_PROC $FILENAME.exo
```

Recent commits to stk should prevent the above issue from manifesting as a subtle bug; instead, you should get an exception reminding you to update your nemesis files.

The second issue is the change in behavior from MeshReadWriteUtils to StkMeshIoBroker with respect to registering Fields for input/output. MeshReadWriteUtils did this automatically and always seemed to pick up the correct fields while StkMeshIoBroker forces the user to explicitly register fields. Neglecting to register a field for input will mean the Field will just contain all zeroes after input. Neglecting to register a field for output will mean the Field won't be in the outputted exodus file. Registering a Field for input that's not in the input file will cause an exception during the read_defined_input_fields method. Re-registering a Field with a different role than the one StkMeshIoBroker expects will cause an exception at the point of registration. A simple way to always get the behavior that was there before is TODO.