M3MS-16OR0401085 - Implement DTK Fortran Interface for FSI Activities

Stuart Slattery
Damien Lebrun-Grandie

September 19, 2016

Approved for public release. Distribution is unlimited.
DOCUMENT AVAILABILITY


Website: http://www.osti.gov/scitech/

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: 703-605-6000 (1-800-553-6847)
TDD: 703-487-4639
Fax: 703-605-6900
E-mail: info@ntis.fedworld.gov
Website: http://www.ntis.gov/help/ordermethods.aspx

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831
Telephone: 865-576-8401
Fax: 865-576-5728
E-mail: report@osti.gov
Website: http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
# CONTENTS

| LIST OF TABLES | v |
| ACRONYMS | vii |
| 1. Introduction | 1 |
| 2. Nek5000/Diablo FSI Algorithm | 3 |
| 3. DTK Solution Transfer Algorithms for FSI | 5 |
| 4. DTK C/Fortran Interfaces | 9 |
| 4.1 C Interface | 9 |
| 4.1.1 DTK_Map Type | 9 |
| 4.1.2 Data Layout | 9 |
| 4.1.3 Map Creation | 9 |
| 4.1.4 Map Application | 11 |
| 4.1.5 Map Destruction | 12 |
| 4.2 Fortran Interface | 12 |
| 4.3 Tests and Examples | 14 |
| 5. Building DTK on ALCF Resources | 15 |
| 5.1 Software environment | 15 |
| 5.2 Configuring Trilinos with DTK | 15 |
| 6. Pseudo-Code Examples | 17 |
| 6.1 Fluid-Side Traction Integration | 18 |
| 6.2 Solid-Side Traction Integration | 19 |
| 7. REFERENCES | 21 |
# LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$DTK_{Map_create}()$ input parameters.</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>$DTK_{Map_apply}()$ input parameters.</td>
<td>12</td>
</tr>
<tr>
<td>ACRONYMS</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ORNL</td>
<td>Oak Ridge National Laboratory</td>
<td></td>
</tr>
<tr>
<td>NEAMS</td>
<td>Nuclear Energy Advanced Modeling and Simulation</td>
<td></td>
</tr>
<tr>
<td>DTK</td>
<td>DataTransferKit</td>
<td></td>
</tr>
<tr>
<td>FSI</td>
<td>Fluid-Structure-Interaction</td>
<td></td>
</tr>
<tr>
<td>FIV</td>
<td>Flow-Induced Vibration</td>
<td></td>
</tr>
<tr>
<td>ALCF</td>
<td>Argonne Leadership Computing Facility</td>
<td></td>
</tr>
</tbody>
</table>
1. Introduction

This report documents the development of DataTransferKit (DTK) [19] C and Fortran interfaces for fluid-structure-interaction (FSI) simulations in NEAMS. In these simulations, the codes Nek5000 [8] and Diablo [13] are being coupled within the SHARP [17] framework to study flow-induced vibration (FIV) in reactor steam generators [12]. We will review the current Nek5000/Diablo coupling algorithm in SHARP and the current state of the solution transfer scheme used in this implementation. We will then present existing DTK algorithms which may be used instead to provide an improvement in both flexibility and scalability of the current SHARP implementation. We will show how these can be used within the current FSI scheme using a new set of interfaces to the algorithms developed by this work. These new interfaces currently expose the mesh-free solution transfer algorithms in DTK, a C++ library, and are written in C and Fortran to enable coupling of both Nek5000 and Diablo in their native Fortran language. They have been compiled and tested on Cooley, the test-bed machine for Mira at ALCF.
2. Nek5000/Diablo FSI Algorithm

The current SHARP FSI algorithm for coupling Nek5000 and Diablo is presented in [12]. In this algorithm, the fluid and structure code are iterated to consistency within a time step. Within an iteration, the solution transfer requirements are the full fluid load field communicated to the structure and an under-relaxed structure displacement field communicated to the fluid at each sub-iteration. In more detail, equation 3 in [12] gives the updated fluid traction field, \( t_{FSI}^{n+1} \), on the structure wetted surface:

\[
t_{FSI}^{n+1} = -p_{n+1}^n + \nu(\nabla v_{n+1}^n) \cdot \mathbf{n},
\]

where \( n \) is the time step, \( p_{n+1}^n \) the fluid pressure, \( \mathbf{n} \) the outward facing normal from the structure surface, \( \nu \) the fluid viscosity, \( v \) the fluid velocity and:

\[
\sigma_f^{n+1} = \nu(\nabla v_{n+1}^n),
\]

the fluid viscous stress tensor. The under-relaxed displacement update of the fluid mesh is given by equation 4 in [12]:

\[
u_{new}^{FSI} = \alpha \nu_{actual}^{FSI} + (1 - \alpha)\nu_{old}^{FSI},
\]

where \( \nu_{actual}^{FSI} \) is the recently computed structure displacement field reconstructed on the fluid grid and \( \alpha \) is the relaxation factor.

In its current state, the solution transfer in the coupling algorithm has 4 key features as given in [12]:

1. Both the fluid and structure mesh are generated so that the interface meshes match node-for-node, creating an “exchange surface”. This makes Eq (1) and Eq (3) easy to satisfy.
2. The fluid and solid meshes are generated separately using the exchange surface as a constraint.
3. A full copy of the exchange surface is kept on each processor in parallel, even though the meshes may be fully distributed.
4. To enable node-to-node communication between codes on the exchange surface a pre-processing step is performed to map between the different set of fluid and solid node IDs on the exchange surface.

The purpose of this work is to incorporate DTK into the FSI scheme within SHARP to enable more efficient and flexible parallel coupling of Nek5000 and Diablo. Using DTK will improve or modify these features in the following ways:

1. DTK allows for coupling between arbitrary surface meshes discretizing the same underlying geometry. Therefore, an “exchange surface” is not needed.
2. Because DTK eliminates the need for an exchange surface, this is additionally eliminated as a meshing constraint and the fluid and structure codes may separately mesh their domains as needed to satisfy accuracy requirements.
3. DTK allows for coupling between fully distributed meshes and maintains and executes a fully distributed communication plan. This will eliminate the global reductions and single processor coupling required by the current exchange surface scheme to improve overall scalability for very large problems.
4. The ID mapping preprocessing step in the current algorithm will be replaced by the generation of a DTK solution transfer map.
3. DTK Solution Transfer Algorithms for FSI

Selection of a solution transfer algorithm for FSI is highly dependent on the physics of the problem being solved. The literature is relatively clear that when transferring displacements from the solid to fluid grid one can choose basic interpolation methods and expect reasonable results for most problems. However, when moving from a one-to-one mapping between the nodes of the structural and fluid grids to a general reconstruction scheme on arbitrary meshes, one must be careful to consider accuracy and conservation of the fluid load solution transfer to avoid large numerical errors and instabilities in the FSI iteration scheme.

In particular, there are two classes of algorithms widely used in the FSI community for reconstruction of fluid load on the solid surface: consistent interpolation schemes and conservative projection schemes. Within these two categories there are further sub-categories of variations to additionally consider.

As a general observation, the literature indicates that in the presence of shocks and other large gradients, conservative projections schemes, while ultimately numerically diffusive, are needed to avoid large errors and ensure conservation when computing fluid load on the solid surface [3, 6]. These schemes are based on variants of weighted residual methods that typically rely on minimizing some norm of a solution transfer problem (typically the L2 norm) subject to some set of constraints. In these algorithms, one constructs a mass matrix and load vector by numerically integrating the basis functions of the fluid and solid grids and then solves the resulting linear system to reconstruct the fluid load on the solid surface. Depending on the means by which the numerical integration is performed the algorithm will have different properties. For example, in transferring load from the fluid to the solid side performing the integration on the fluid side will yield better conservation while integrating on the solid side will yield better accuracy. In [11], it was found that using an integration grid consisting of an intersection between the fluid and structure sides yields both good conservation and accuracy properties as demonstrated on a number of problems [9, 10]. Similar conservative algorithms, such as the super-mesh approach given in [7] have been proposed as well to similar effect.

If the fluid pressure and viscous stress fields are reasonably smooth, the results in [6] indicate that conservative projection schemes are not needed in the load computation. Instead, consistent interpolations may be used satisfactorily. This is particularly advantageous because these algorithms are typically faster, easier to implement, and require less information about the grids and discretization of the fluid and structure problems. There are two choices for composing the load vector when using consistent interpolation. The first choice is to integrate the traction on the fluid surface:

$$f_f = \int_{\Gamma_f} (\sigma_f \cdot n - p n) d\Gamma_f,$$

and then transfer from the fluid nodes to the structural nodes with a map operator, $M$:

$$f_s = M f_f.$$  \hspace{1cm} (5)

The transfer operator in this case can then be used to transfer the structure displacements back to the fluid grid to satisfy conservation of virtual work. Virtual work is defined as the virtual displacement multiplied by the forces and must be balanced on the fluid and solid side of the interface such that:

$$\delta W = \delta u_f^T \cdot f_f = \delta u_s^T \cdot f_s.$$  \hspace{1cm} (6)

Substituting in Eq (5) and taking the transpose we get:

$$\delta u_f = M \delta u_s^T,$$  \hspace{1cm} (7)
which is then be used to formulate the transfer of displacements from the structure to the fluid using the transpose of the original map operator:

\[ \mathbf{u}_f = \mathbf{M}^T \mathbf{u}_s. \]  

(8)

The second choice is to transfer the fluid pressure and viscous shear stresses to quadrature points on the structure surface and integrate the traction on the structure. Here, we use interpolation to develop some functional approximation to the fluid pressures and stresses which may be evaluated at locations on the fluid surface, \( \mathbf{x}_s \), such that:

\[ \mathbf{f}_s = \int_{\Gamma_s} (\sigma_f(\mathbf{x}_s) \cdot \mathbf{n} - p(\mathbf{x}_s)\mathbf{n})d\Gamma_s. \]  

(9)

If we numerically integrate this function to compose the load on the structural nodes:

\[ \mathbf{f}_s = \sum_i w_i \phi_i(\sigma_f(\mathbf{x}_i) \cdot \mathbf{n}(\mathbf{x}_i) - p(\mathbf{x}_i)\mathbf{n}(\mathbf{x}_i)) \]  

(10)

we see that the task is to now reconstruct the fluid pressure and viscous shear stress fields on the quadrature points of the structural mesh. Here, \( w_i \) are the quadrature point weights, \( \phi_i \) the value of the shape function values, \( \mathbf{x}_i \) the physical location of the integration points, and the shear stress, pressure, and surface normal fields are all evaluated at the quadrature points in the physical frame. We can perform the reconstruction on the quadrature points in an equivalent manner to Eq (5) using a fluid node to solid quadrature point map, \( \mathbf{M}_{fs} \):

\[ p_s = \mathbf{M}_{fs} p, \]  

(11)

and

\[ \sigma_{fs} = \mathbf{M}_{fs} \sigma_f, \]  

(12)

with the \( i^{th} \) components of the vectors correlating to quadrature point such that \( p_{s, i} = p(\mathbf{x}_i) \) and \( \sigma_{fs, i} = \sigma_f(\mathbf{x}_i) \). For the displacements we use the same technique as in Eq (8) by using a structure node to fluid node map, \( \mathbf{M}_{sf} \):

\[ \mathbf{u}_f = \mathbf{M}_{sf} \mathbf{u}_s. \]  

(13)

When a mesh is available, basic shape function interpolations can be used to compose the map operators for consistent interpolation to build a representation of the nodal fluid quantities on the structural grid. As an alternative, in the last 20 years a number of new mesh-free algorithms based on point clouds and radial basis functions have been developed for consistent interpolation in FSI problems [2, 5, 15, 16, 14, 1, 4]. In these algorithms, the underlying support of the solid and fluid surfaces is replaced by a point cloud representation of the surfaces with these points either consisting of the mesh nodes or quadrature points depending on where the data is to be interpolated. Several of these mesh-free algorithms have been implemented in DTK, studied for several problem variations, and developed to be used for very large parallel problems [18]. In particular, three algorithms are currently available:

1. **Node-To-Node Mapping**: A one-to-one map between fluid and structure nodes was specifically developed for this work to provide an exact numerical replacement for the existing functionality described in [12]. This implementation can serve as a starting point for testing the integration of DTK into the coupled application. In addition, the map uses DTK parallel search algorithms such that the Diablo and Nek5000 need only to provide local mesh and field data. This map will only work in cases of matching fluid and structure meshes but the meshes may be arbitrarily decomposed in parallel.
2. **Spline Interpolation**: An implementation of the algorithm developed by Beckert and Wendland in [2]. This algorithm has a reasonably small setup cost but requires the solution of a global, albeit sparse, saddle-point problem using a Krylov solver at every application of the map and therefore is more expensive. This map will work for non-matching meshes of arbitrary parallel decomposition.

3. **Moving Least Square Reconstruction**: An implementation of the algorithm initially developed by Quaranta, Maserati, and Mantegazza in [14]. This algorithm has a larger setup cost as documented in [18], but the application of the operator is extremely inexpensive. This map will also work for non-matching meshes of arbitrary parallel decomposition.

The node-to-node mapping and the moving least square reconstruction may be used for the virtual work conserving consistent interpolation scheme described above due to the availability of a transpose operation while all three algorithms may be used in the scheme where pressures and stresses are projected to the structure quadrature points. A transpose operation is currently not available for the spline interpolation operator.
4. DTK C/Fortran Interfaces

Much of this work has been dedicated to developing C and Fortran interfaces to the mesh-free algorithms in DTK to allow coupling of Diablo and Nek5000 in their native Fortran language. The interface is broken up into three pieces: map creation, map application, and map destruction.

4.1 C Interface

First we provide the C interface to the mesh-free operators. The Fortran interfaces will effectively wrap around these C interfaces to provide identical functionality.

4.1.1 DTK_Map Type

The C interface defines an opaque handle to $\text{DTK\_Map}$ data type which encapsulates the underlying C++ data structures in DTK.

\[ \text{typedef void DTK\_Map;} \]

All DTK services will be accessed through this handle.

4.1.2 Data Layout

The layout of data arrays provided to the interface is enumerated:

\[ \text{typedef enum data\_layout \{ DTK\_BLOCKED, DTK\_INTERLEAVED \}} \]

where a blocked layout indicates that data is blocked by dimension and interleaved indicates that the dimensions are interleaved. As an example, consider an array $x$ which has $n$ elements and three dimensions. In blocked format the data layout is:

\[
x = \begin{bmatrix}
x_{1,1} & x_{2,1} & \ldots & x_{n,1} \\
x_{1,2} & x_{2,2} & \ldots & x_{n,2} \\
\end{bmatrix},
\]

and in interleaved format the data layout is

\[
x = \begin{bmatrix}
x_{1,1} & x_{1,2} & x_{1,3} & x_{2,1} & x_{2,2} & x_{2,3} & \ldots & x_{n,1} & x_{n,2} & x_{n,3}
\end{bmatrix}.
\]

Another way to consider data layout is right and left ordering. In the blocked case, the element index changes the fastest and the dimension index changes the slowest while interleaved is reversed. In addition, accepting data as flat arrays with an indication of data layout allows codes to provide data in an ordering that is best for their data structures and programming language of choice.

4.1.3 Map Creation

Map creation allocates all DTK data structures and generates the mapping between the input point clouds with the following API:
DTK_Map* DTK_Map_create( MPI_Comm comm,  
double const* src_coord,  
unsigned src_num,  
DTK_Data_layout src_layout,  
double const* tgt_coord,  
unsigned tgt_num,  
DTK_Data_layout tgt_layout,  
int space_dim,  
char const* options);

which returns a pointer to an allocated DTK_Map object. The user is responsible for destroying this object by calling DTK_Map_delete(). The input arguments for DTK_Map_create() are defined in the table below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm</td>
<td>The MPI communicator on which to build the map.</td>
</tr>
<tr>
<td>src_coord</td>
<td>Pointer to the local coordinates of the point cloud acting as the data source.</td>
</tr>
<tr>
<td>src_num</td>
<td>The number of points in the source point cloud given in src_coord.</td>
</tr>
<tr>
<td>src_layout</td>
<td>The data layout of the coordinates src_coord organized by point and spatial dimension.</td>
</tr>
<tr>
<td>tgt_coord</td>
<td>Pointer to the local coordinates of the point cloud acting as the data target.</td>
</tr>
<tr>
<td>tgt_num</td>
<td>The number of points in the target point cloud given in tgt_coord.</td>
</tr>
<tr>
<td>tgt_layout</td>
<td>The data layout of the coordinates in tgt_coord organized by point and spatial dimension.</td>
</tr>
<tr>
<td>space_dim</td>
<td>The spatial dimension of the problem.</td>
</tr>
<tr>
<td>options</td>
<td>Character string of map options including map type, parallel search type, and basis type.</td>
</tr>
</tbody>
</table>

Options are provided using a string in the JSON format. An example of options using for map creation are:

```plaintext
char options[] = "{ "\n"\"Map Type\": \"Moving Least Square Reconstruction\", "\n"\"Basis Type\": \"Wendland\", "\n"\"Basis Order\": 2, "\n"\"Search Type\": \"Radius\", "\n"\"RBF Radius\": 0.3 \}";
```

which would create a moving least square reconstruction map using a Wendland radial basis function of order 2 using a radial search with a radius of 0.3 for support. As another example:
The code snippet provided creates a spline interpolation operator using a Wu basis of order 4 with a nearest neighbor support group of 11 points for each interpolation. A set of reasonable default options is provided. The following are the available options:

- **Available values for Map Type** are: **Node To Node**, **Moving Least Square Reconstruction**, and **Spline Interpolation**
- **Available values for Basis Type** and **Basis Order** are:
  - Wendland with orders 0, 2, 4, and 6
  - Wu with orders 2 and 4
  - Buhmann with order 3
- **Available values for Search Type** are:
  - Radius uses a radial search to find nearest neighbors in a point cloud and sets the effective radius of the basis functions. When using this search type, set the parameter **RBF Radius** to any floating point number greater than zero to define the search radius.
  - Nearest Neighbor will use a search that finds the k-nearest neighbors in a point cloud. The distance of the furthest neighbor sets the effective radius for the basis functions. When using this search type, set the parameter **Num Neighbors** to any integer greater than zero to define the number of neighbors used support any given point.

Search and basis parameters are not used by the node-to-node mapping.

### 4.1.4 Map Application

After creation, the map operator may be applied as many times as needed to map different data sets between the two sets of input point clouds. If those point clouds change, then the map will need to be regenerated to represent the addition, removal, or movement of points in the clouds. The mapping is applied with the following API:

```c
void DTK_Map_apply( DTK_Map* dtk_map,
                     double const* src_field,
                     DTK_Data_layout src_layout,
                     double* tgt_field,
                     DTK_Data_layout tgt_layout,
                     int field_dim,
                     bool transpose );
```
The input arguments for `DTK_Map_apply()` are defined in the table below.

**Table 2. `DTK_Map_apply()` input parameters.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtk_map</code></td>
<td>The DTK map to apply.</td>
</tr>
<tr>
<td><code>src_field</code></td>
<td>Pointer to data on the source point cloud. Must have the same number and order of points as in map creation.</td>
</tr>
<tr>
<td><code>src_layout</code></td>
<td>The data layout of the data in <code>src_field</code> organized by point and field dimension.</td>
</tr>
<tr>
<td><code>tgt_field</code></td>
<td>Pointer to data on the target point cloud. Must have the same number and order of points as in map creation.</td>
</tr>
<tr>
<td><code>tgt_layout</code></td>
<td>The data layout of the data in <code>tgt_field</code> organized by point and field dimension.</td>
</tr>
<tr>
<td><code>field_dim</code></td>
<td>The dimension of the field being transferred.</td>
</tr>
<tr>
<td><code>transpose</code></td>
<td>Transpose apply option. If true apply the transpose of the map.</td>
</tr>
</tbody>
</table>

### 4.1.5 Map Destruction

Users are responsible for destroying all DTK maps they have created by calling `DTK_Map_delete()`:

```c
void DTK_Map_delete( DTK_Map* dtk_map );
```

After deleting the map, the pointer becomes null and the map may no longer be used. Failure to destroy a map after creation will result in a memory leak.

### 4.2 Fortran Interface

A Fortran interface was developed to provide bindings for the C implementation. This includes the data layout enumeration:

```fortran
enum, bind(C)
    enumerator :: DTK_BLOCKED=1, DTK_INTERLEAVED=2
end enum
```
Map creation:

```fortran
function DTK_Map_create(comm, src_coord, src_num, src_layout, &
tgt_coord, tgt_num, tgt_layout, space_dim, options) &
result(dtk_map) &
  bind(C, name="DTK_Map_create_f")
use iso_c_binding
implicit none
type(c_ptr) :: dtk_map
integer(kind=c_int), value :: comm
integer(kind=c_size_t), value :: src_num
integer(kind=c_int), value :: src_layout
integer(kind=c_size_t), value :: tgt_num
integer(kind=c_int), value :: tgt_layout
integer(kind=c_int), value :: space_dim
character(kind=c_char) :: options(*)
end function DTK_Map_create
```

Please note that the string options must be NULL-terminated to satisfy C string handling requirements. The Fortran version of the first example of options provided in the previous section would be:

```fortran
options = '{
  &"Map Type": "Moving Least Square Reconstruction", &
  &"Basis Type": "Wendland", &
  &"Basis Order": 2, &
  &"RBF Radius": 0.3 }'//c_null_char
```

Map application:

```fortran
subroutine DTK_Map_apply(dtk_map, src_field, src_layout, &
tgt_field, tgt_layout, field_dim, apply_transpose) &
  bind(C, name="DTK_Map_apply")
use iso_c_binding
implicit none
type(c_ptr), value :: dtk_map
type(c_ptr), value :: src_field
integer(kind=c_int), value :: src_layout
type(c_ptr), value :: tgt_field
integer(kind=c_int), value :: tgt_layout
integer(kind=c_int), value :: field_dim
logical(kind=c_bool), value :: apply_transpose
end subroutine DTK_Map_apply
```
Map destruction:

```fortran
subroutine DTK_Map_delete(dtk_map) 
   bind(C, name="DTK_Map_delete")
   use iso_c_binding
   implicit none
   type(c_ptr), value :: dtk_map
end subroutine DTK_Map_delete
```

In all cases, the function arguments have the same structure as those outlined in the C functions in § 4.1. The two exceptions are for map creation in `DTK_Map_create` where an integer is passed to represent the MPI handle and the null-termination requirement for the options string.

### 4.3 Tests and Examples

Unit tests for the work presented in this report and integration tests that can be used as examples for the interfaces can be found in the following files. The repo hash `f28542b02893d0f1087258c1d3b2763b41392ac` was used as a reference.

- **In DataTransferKit/packages/Adapters/C_API/test:**
  - `tstPOD_PointCloudEntity.cpp`, `tstPOD_PointCloudEntityIterator.cpp`, `tstPOD_PointCloudEntitySet.cpp`, and `tstPOD_PointCloudEntityLocalMap.cpp`
  - Test basic DTK adapters for POD data structures
  - `tstPOD_C_API.cpp` tests and serves as an example of using the new C interface from C++ code
  - `tstC_API.cpp` tests and serves as an example of using the new C interface from C code

- **In DataTransferKit/packages/Adapters/Fortran_API/test:**
  - `tstFortran_API.F90` tests and serves as an example of using the new Fortran interface from Fortran 90 code

- **In DataTransferKit/packages/Operators/test:**
  - `tstNodeToNodeOperator.cpp` tests the node-to-node mapping added for this work as a drop-in replacement for existing Nek5000/Diablo coupling functionality
  - `tstVirtualWork.cpp` tests the conservation of virtual work as given by Eq (6) using the moving least square reconstruction operator
5. Building DTK on ALCF Resources

DTK and its new C/Fortran interfaces were built and tested on Cooley at ALCF which has the same software environment and network as Mira.

5.1 Software environment

The MVAPICH2 wrappers of the Intel Composer XE compilers 2016 were used to build DTK. Below is the corresponding SoftEnv configuration file. $HOME/.soft.cooley.

```bash
+mvapich2-intel
+intel-composer-xe-2016
+gcc-4.8.1
@default
```

Note that without the +gcc-4.8.1, compile time errors with `std::unordered_map::emplace(...)` prevent DTK to build due to an older default compiler version. By default, CMake points to the GNU implementation of the Standard Template Library and the default 4.4.7 version does not have full C++11 support.

We were able to use the Intel Math Kernel Library (MKL) implementation for the BLAS/LAPACK and the 1.57.0 version of Boost that was available on the ALCF machines.

5.2 Configuring Trilinos with DTK

Here is an example of configuration script to build DTK and its unit tests on Cooley using Trilinos version 12.4.2.

```bash
#!/usr/bin/env bash
EXTRA_ARGS=$@
make \n  D CMAKE_BUILD_TYPE=Release \n  D TPL_ENABLE_MPI=ON \n  D TPL_ENABLE_BLAS=ON \n  D TPL_ENABLE_LAPACK=ON \n  D TPL_ENABLE_Boost=ON \n  D TPL_ENABLE_BoostLib=ON \n  D TPL_ENABLE_Netcdf=OFF \n  D TPL_ENABLE_MOAB=OFF \n  D TPL_ENABLE_Libmesh=OFF \n  D Trilinos_ENABLE_CXX11=ON \n  D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF \n  D Trilinos_ENABLE_DATA_TRANSFER=ON \n  D DataTransferKit_ENABLE_DBBC=OFF \n  D DataTransferKit_ENABLE_TESTS=ON \n
<path to Trilinos source dir>
```
6. Pseudo-Code Examples

The following are pseudo-code examples illustrating the use of the C interface for both fluid-side and solid-side variations of creating the load vector.
6.1 Fluid-Side Traction Integration

```c
// Get the parallel communicator. Does not have to be comm world.
MPI_Comm comm = MPI_COMM_WORLD;

// Get the spatial dimension.
const int space_dim = 3;

// Get the number of nodes in the fluid surface mesh.
const unsigned num_fluid_nodes = fluid_solver.numNodes();

// Get the nodal coordinates of the fluid surface mesh.
const double* fluid_coords = fluid_solver.getNodeCoords();

// Fluid data is blocked.
DTK_Data_layout fluid_layout = DTK_BLOCKED;

// Get the number of nodes in the structure surface mesh.
const unsigned num_structure_nodes = structure_solver.numNodes();

// Get the nodal coordinates of the structure surface mesh.
const double* structure_coords = structure_solver.getNodeCoords();

// Structure data is interleaved.
DTK_Data_layout structure_layout = DTK_INTERLEAVED;

// Create a map to transfer from the structure nodes to the fluid nodes using
// the default options. We will use the transpose to transfer in the opposite
// direction to satisfy conservation of virtual work.
DTK_Map* structure_to_fluid_map = DTK_Map_create( comm, fluid_coords, num_fluid_nodes, fluid_layout, structure_coords, num_structure_nodes, structure_layout, space_dim );

// Time integration loop.
for ( int t = 0; t < num_time_steps; ++t )
{
    // Solve the structural problem.
    structure_solver.solve();

    // Get the displacements from the structure.
    const double* structure_displacements = structure_solver.getDisplacements();

    // Get the fluid mesh displacement array to write into.
    double* fluid_displacements = fluid_solver.getDisplacements();

    // Transfer the structure displacements to the fluid solver.
    DTK_Map_apply( structure_to_fluid_map, structure_displacements, structure_layout, fluid_displacements, fluid_layout, space_dim, false );

    // Solve the fluid problem.
    fluid_solver.solve();

    // Get the integrated load from the fluid solver.
    const double* fluid_load = fluid_solver.getLoad();

    // Get the load array from the structure solver to write into.
    double* structure_load = structure_solver.getLoad();

    // Apply the transpose of the map to transfer the loads from the fluid
    // mesh to the structure mesh to conserve virtual work.
    DTK_Map_apply( structure_to_fluid_map, fluid_load, fluid_layout, structure_load, structure_layout, space_dim, true );
}

// Delete the map after time integration has finished.
DTK_Map_delete( structure_to_fluid_map );
```
6.2 Solid-Side Traction Integration

// Get the parallel communicator. Does not have to be comm world.
MPI_Comm comm = MPI_COMM_WORLD;

// Set the spatial dimension.
const int space_dim = 3;

// Get the nodes in the fluid surface mesh.
const unsigned num_fluid_nodes = fluid_solver.numNodes();
const double* fluid_coords = fluid_solver.getNodeCoords();

// Fluid data is blocked.
DTK_Data_layout fluid_layout = DTK_BLOCKED;

// Get the nodes in the structure surface mesh.
const unsigned num_structure_nodes = structure_solver.numNodes();
const double* structure_node_coords = structure_solver.getNodeCoords();

// Get the quadrature points in the structure surface mesh.
const unsigned num_structure_quad_points = structure_solver.numQuadPoints();
const double* structure_quad_points = structure_solver.getQuadPointCoords();

// Structure data is interleaved.
DTK_Data_layout structure_layout = DTK_INTERLEAVED;

// Create a map to transfer from the structure nodes to the fluid nodes using
// the default options.
DTK_Map* structure_to_fluid_map = DTK_Map_create( comm,
fluid_coords,
num_fluid_nodes,
fluid_layout,
structure_node_coords,
num_structure_nodes,
structure_layout,
space_dim );

// Create a map to transfer from the fluid nodes to the structure quadrature
// points using the default options.
DTK_Map* fluid_to_structure_map = DTK_Map_create( comm,
fluid_coords,
num_fluid_nodes,
fluid_layout,
structure_quad_point_coords,
num_structure_quad_points,
structure_layout,
space_dim );

// Time integration loop.
for ( int t = 0; t < num_time_steps; ++t )
{
    // Solve the structural problem.
    structure_solver.solve();

    // Get the displacements from the structure.
    const double* structure_displacements = structure_solver.getDisplacements();

    // Get the fluid mesh displacement array to write into.
    double* fluid_displacements =
        fluid_solver.getDisplacements();

    // Transfer the structure displacements to the fluid solver.
    DTK_Map_apply( structure_to_fluid_map,
structure_displacements,
structure_layout,
fluid_displacements,
false );

    // Solve the fluid problem.
    fluid_solver.solve();

    // Get the pressures from the fluid solver.
    const double* fluid_pressure = fluid_solver.getPressure();

    // Get the pressure array from the structure solver to write into.
    double* structure_pressure = structure_solver.getPressure();

    // Get the viscous shear stresses from the fluid solver.
    const double* fluid_stress = fluid_solver.getViscousShearStress();

    // Get the viscous shear stress array from the structure solver to write
    // into.
    double* structure_stress = structure_solver.getViscousShearStress();
// Transfer the pressure from the fluid nodes to structure quadrature points.
DTK_Map_apply( fluid_to_structure_map
fluid_pressure,
fluid_layout,
structure_pressure,
structure_layout,
1,
false );

// Transfer the viscous shear stresses from the fluid nodes to structure quadrature points.
DTK_Map_apply( fluid_to_structure_map
fluid_stress,
fluid_layout,
structure_stress,
structure_layout,
6,
false );

// Delete the maps after time integration has finished.
DTK_Map_delete( fluid_to_structure_map );
DTK_Map_delete( structure_to_fluid_map );
7. REFERENCES


