DataTransferKit Documentation

Release 3.0.0

Stuart Slattery, Damien Lebrun-Grandie, Bruno Turcksin, Andrey P

Oct 27, 2018

Contents

Contents:

CHAPTER 1

Getting started with DTK

1.1 Overview

DataTransferKit is an open-source software library of parallel solution transfer services for multiphysics simulations. DTK uses a general operator design to provide scalable algorithms for solution transfer between shared volumes and surfaces.

DTK was originally developed at the University of Wisconsin - Madison as part of the Computational Nuclear Engineering Group (CNERG) and is now actively developed at the Oak Ridge National Laboratory as part of the Computational Engineering and Energy Sciences (CEES) group.

DTK is supported and used by the following projects and programs:

- Exascale Computing Project (ECP ALExa)
- Oak Ridge National Laboratory (ORNL) Laboratory Directed Research and Development (LDRD)
- Consortium for Advanced Simulation of Light Water Reactors (CASL)
- Nuclear Energy Advanced Modeling and Simulation (NEAMS)
- National Highway Traffic Safety Administration (NHTSA)

1.2 DataTransferKit Development Team

DTK is developed and maintained by:

- Stuart Slattery
- Damien Lebrun-Grandie
- Bruno Turcksin
- Andrey Prokopenko

Alumni:

- Roger Pawlowski
- Alex McCaskey

1.3 DataTransferKit Packages

DTK has the following packages:

Utils General utilities for software development including exception handling, and other functional programming tools

Interface Interfaces with user applications and operators

Search Search algorithms leveraged by all operators

Meshfree Point cloud based operators (e.g., nearest neighbor, moving least squares, spline interpolation)

Discretization Mesh based operators (e.g., interpolation, L2 projection)

Benchmarks Mesh and partitioning infrastructure of problems relevant to DTK

MapFactory Map operators used by the C interface

1.4 Questions, Bug Reporting, and Issue Tracking

Questions, bug reporting and issue tracking are provided by GitHub. Please report all bugs by creating a new issue. You can ask questions by creating a new issue with the question tag.

1.5 Publications

Publications to date related to DataTransferKit:

- S. Slattery, *"Mesh-Free Data Transfer Algorithms for Partitioned Multiphysics Problems: Conservation, Accuracy, and Parallelism"*, Journal of Computational Physics, vol. 307, pp. 164-188, 2016.
- S. Slattery, S. Hamilton, T. Evans, *"A Modified Moving Least Square Algorithm for Solution Transfer on a Spacer Grid Surface"*, ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method, Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015).
- R. Schmidt, K. Belcourt, R. Hooper, R. Pawlowski, K. Clarno, S. Simunovic, S. Slattery, J. Turner, S. Palmtag, *"An Approach for Coupled-Code Multiphysics Core Simulations from a Common *Input"*, Annals of Nuclear Energy, Volume 84, pp. 140-152, 2014.
- S. Slattery, P.P.H. Wilson, R. Pawlowski, *"The Data Transfer Kit: A Geometric Rendezvous-Based Tool for Multiphysics Data Transfer"*, International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), American Nuclear Society, Sun Valley, ID, May 5-9, 2013.

CHAPTER 2

Installation

This section provide guidelines for installing DataTransferKit and its TPLs.

2.1 Install third-party libraries

The following third party libraries (TPLs) are used by DTK:

Packages	Dependency	Version
Boost	Required	1.61.0
BLAS/LAPACK	Required	N/A
MPI	Required	N/A
Trilinos	Required	12.14
Google Benchmark	Required (if examples are enabled)	1.4

Note: DTK is built as an external package of Trilinos. Thus, DTK needs the source of the Trilinos library rather than an installed version.

The dependencies of DataTransferKit may be built using Spack package manager. You need to install the following packages:

```
$ spack install openblas
$ spack install boost
$ spack install mpi
$ spack install benchmark
```

Once installed, the module files for the packages must be loaded into the environment by doing

```
$ spack load openblas
$ spack load boost
```

(continues on next page)

(continued from previous page)

```
$ spack load openmpi
$ spack load benchmark
```

2.2 DTKData repository

The DTKData repository contains mesh files used in DTK examples. To build the examples and include the mesh files include the git submodule with your cloned repository:

```
$ git submodule init
$ git submodule update
```

Another way to achieve this is to pass the `--recursive` option to the `git clone` command which will automatically initialize and update DTKData in the DataTransferKit repository.

You can also download the DTKData repository yourself and then, link it into DTK directory:

```
$ cd $DTK_DIR
$ ln -s $DTKDATA_DIR DTKData
```

2.3 Building DTK

DTK is configured and built using TriBITS. DTK builds within Trilinos effectively as an extension package. First, link DTK into the Trilinos main directory:

```
$ cd $TRILINOS_DIR
$ ln -s $DTK_DIR DataTransferKit
```

Create a `do-configure` script such as:

```
EXTRA_ARGS=$@

cmake \
    -D CMAKE_BUILD_TYPE=Release \
    -D TPL_ENABLE_MPI=ON \
    -D TPL_ENABLE_BLAS=ON \
    -D TPL_ENABLE_LAPACK=ON \
    -D TPL_ENABLE_Boost=ON \
    -D Trilinos_ENABLE_EXPLICIT_INSTANTIATION=ON \
    -D Tpetra_INST_INT_LONG_LONG=OFF \
    -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF \
    -D Trilinos_EXTRA_REPOSITORIES="DataTransferKit" \
    -D Trilinos_ENABLE_DataTransferKit=ON \
    -D DataTransferKit_ENABLE_DBC=ON \
    -D DataTransferKit_ENABLE_TESTS=ON \
    -D DataTransferKit_ENABLE_EXAMPLES=ON \
    $EXTRA_ARGS \
    $TRILINOS_DIR
```

and run it from your build directory:

```
$ mkdir build && cd build
$ ../do-configure
```

More install scripts can be found in `scripts/`.

Note: DTK has only been tested with CUDA 8.0. Other versions of CUDA may or may not work.

Note: If DBC is OFF some tests will fail.

2.4 Build this documentation

Building documentation requires sphinx. (Re)configure with `-D DataTransferKit_ENABLE_ReadTheDocs=ON` and run:

```
$ make docs
```

Open the `index.html` in the directory `DataTransferKit/docs/html`.

2.5 Generate Doxygen documentation

Configure with `-D DataTransferKit_ENABLE_Doxygen=ON` and run:

```
$ make doxygen
```

Checkout `DataTransferKit/docs/doxygen/html/index.html`.

CHAPTER 3

DataTransferKit API

3.1 C API

Typedefs

typedef struct _DTK_UserApplicationHandle ***DTK_UserApplicationHandle**

DTK user application handle.

The user application handle represents an instance of the data access interface to a user application with user implementations of DTK callback functions associated with each individual handle. As many handles may be created as desired with each representing its own unique instance.

Note Note the use of this handle in many interface functions below - all DTK functions that need access to user inputs and outputs will have user application handles as arguments.

typedef struct _DTK_MapHandle ***DTK_MapHandle**

DTK map handle.

The map handle represents a unique instance of a DTK transfer operator. A DTK map transfers data between a source (the application providing the data) and a target (the application receiving the data), each represented by their own user application handle providing access to their geometry, mesh, and field data. The type of map represented by the handle is defined via a set of input options and the execution space where map computations occur is defined at the time of construction via an execution space enumeration.

As many map instances may be created as needed and each instance may represent a different type of transfer operator. Once created, a map instance may be applied to transfer between the source and target as many times as needed as long as the source and target user application handles remain valid.

typedef void (***DTK_NodeListSizeFunction**)(void *user_data, unsigned *space_dim, size_t *local_num_nodes)

Prototype function to get the size parameters for building a node list.

A node list is a collection of spatial points of a given dimension.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_NODE_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension of the node coordinates.
- `local_num_nodes`: Number of nodes DTK will allocate memory for.

typedef void (***DTK_NodeListDataFunction**)(void *user_data, Coordinate *coordinates)

Prototype function to get the data for a node list.

A node is defined by its spatial coordinates.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_NODE_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates. The length of this array is space_dim * local_num_nodes. Coordinates are blocked by dimension. For example, in 3 dimensions the x coordinates for all nodes are listed first followed by all of the y coordinates and then all of the z coordinates. A loop for filling this array in the proper order, for example, would look like:

```
for ( int n = 0; n < local_num_nodes; ++n )
    for ( int d = 0; d < space_dim; ++d )
        coordinates[ d*local_num_nodes + n ] =
            coordinate_of_node_n_in_dimension_d;
```

typedef void (***DTK_BoundingVolumeListSizeFunction**)(void *user_data, unsigned *space_dim, size_t *local_num_volumes)

Prototype function to get the size parameters for building a bounding volume list.

A bounding volume list is a collection of axis-aligned Cartesian boxes in a given spatial dimension.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_BOUNDING_VOLUME_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_volumes`: Number of volumes DTK will allocate memory for.

typedef void (***DTK_BoundingVolumeListDataFunction**)(void *user_data, Coordinate *bounding_volumes)

Prototype function to get the data for a bounding volume list.

A bounding volume is defined by the low and high corner of the box (e.g. [x_min,y_min,z_min] and [x_max,y_max,z_max] in 3 dimensions).

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_BOUNDING_VOLUME_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `bounding_volumes`: Bounding volumes. The length of this array is 2 * space_dim * local_num volumes - enough space for the low and high corner coordinates of each volume. This array specifies the coordinates of the low and high corners of each volume. The array is blocked by corner and the coordinates for each corner are blocked by dimension. The low corner comes first and the high corner comes second. A loop for filling this array in the right order, for example, would look like:

```
for ( int v = 0; v < local_num_volume; ++v )
    for ( int d = 0; d < space_dim; ++d )
    {
        bounding_volumes[ d*local_num_volumes + v ] =
            low_corner_of_volume_v_in_dimension_d;

        bounding_volumes[ (space_dim + d)*local_num_volumes + v ] =
            high_corner_of_volume_v_in_dimension_d;
    }
```

typedef void (***DTK_PolyhedronListSizeFunction**)(void *user_data, unsigned *space_dim, size_t *local_num_nodes, size_t *local_num_faces, size_t *total_face_nodes, size_t *local_num_cells, size_t *total_cell_faces)

Prototype function to get the size parameters for building a polyhedron list.

A polyehdron list is a collection of arbitrary linear polyhedra defined by a set of nodes, faces constructed by ordered loops of nodes, and cells constructed by lists of faces with orientations.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_POLYHEDRON_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_nodes`: Number of nodes DTK will allocate memory for. This is the total number of unique nodes needed to compose all polyhedra in the list on this MPI rank.
- `local_num_faces`: Number of faces DTK will allocate memory for. This is the total number of unique faces needed to compose all polyhedra in the list on this MPI rank.
- `total_face_nodes`: Total number of nodes for all faces. This is equivalent to counting the number of nodes that construct each face and then summing this value over all faces on this MPI rank.
- `local_num_cells`: Number of cells DTK will allocate memory for on this MPI rank.
- `total_cell_faces`: Total number of faces for all cells. This is equivalent to counting the number of faces that construct each cell and then summing this value over all cells on this MPI rank.

typedef void (***DTK_PolyhedronListDataFunction**)(void *user_data, Coordinate *coordinates, LocalOrdinal *faces, unsigned *nodes_per_face, LocalOrdinal *cells, unsigned *faces_per_cell, int *face_orientation)

Prototype function to get the data for a polyhedron list.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_POLYHEDRON_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates. The length of this array is local_num_nodes * space_dim. This array is blocked by dimension in identical fashion to the NodeList coordinates.
- `faces`: Connectivity list of faces. The length of this array is total_face_nodes. This array is defined as rank-1 but represents unstructured rank-2 data. It should be sized as total sum of the number of nodes composing each face or, equivalently, the sum of all elements in the following array, nodes_per_face, which indicates how many nodes are assigned to each face and how to index into this array. The input should be arranged as follows. Consider the n^th node of face i to be f_n^i which is equal to the local index of the corresponding node in the coordinates array. Two faces, the first with 4 nodes and the second with 3 would then be defined via this array as: $(f_1^1, f_2^1, f_3^1, f_4^1, f_1^2, f_2^2, f_3^2)$ with the nodes_per_face array reading $(4, 3)$
- `nodes_per_face`: Number of nodes per face. The length of this array is local_num_faces. For every face, list how many nodes construct it. The sum of all local elements in this array should equal the total size of the faces array.
- `cells`: Connectivity list of polyhedrons. The length of this array is total_cell_faces. This array is defined as rank-1 but represents unstructured rank-2 data. It should be sized as (total sum of the number of faces composing each polyhedron) or the sum of all elements in the array faces_per_cells, which indicates how many faces are assigned to each cell and how to index into this array. The input should be arranged as follows. Consider the n^th face of cell i to be c_n^i which is equal to the local index of the corresponding face in the faces array. Two cells, the first with 5 faces and the second with 4 would then be defined via this array as: $(c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_1^2, c_2^2, c_3^2, c_4^2)$ with the faces_per_cell array reading $(5, 4)$.
- `faces_per_cell`: Number of faces per cell. The length of this array is local_num_cells. For every cell, list how many faces construct it. The sum of all local elements in this view should equal the total size of the cells view. This view is rank-1 and of length of the number of cells in the list.
- `face_orientation`: Orientation of the faces. The length of this array is total_cell_faces. Orientation of each face composing a cell indicating an outward or inward facing normal based on node ordering of the face and use of the right-hand rule. This view is defined as rank-1 but represents unstructured rank-2 data. This view is the same size as the cells view and is indexed in an identical matter. If the face for the given cell has a node ordering that returns a face normal that points into the cell via the right hand rule then a -1 should be input. If the node ordering of the face produces a normal that points out from the cell a +1 should be input.

typedef void (***DTK_CellListSizeFunction**) (void *user_data, unsigned *space_dim, size_t *local_num_nodes, size_t *local_num_cells, size_t *total_cell_nodes)

Prototype function to get the size parameters for building a cell list.

Cells are objects from a topological zoo of cell types (e.g. hexahedron, triangle, etc.) and are defined by a cell type and a set of nodes ordered as prescribed by the cell type. Valid cell topologies are defined in DTK_CellTypes.h

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_CELL_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `space_dim`: Spatial dimension.
- `local_num_nodes`: Number of nodes DTK will allocate memory for.

- `local_num_cells`: Number of cells DTK will allocate memory for.
- `total_cell_nodes`: Total number of nodes for all cells.

typedef void (***DTK_CellListDataFunction**)(void *user_data, Coordinate *coordinates, LocalOrdinal *cells, DTK_CellTopology *cell_topologies)

Prototype function to get the data for a mixed topology cell list.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_CELL_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `coordinates`: Node coordinates. The length of this array is space_dim * local_num_nodes. This array is blocked by dimension in identical fashion to the NodeList coordinates.
- `cells`: List of cells. It represents a lists of cells with different topologies ordered in blocks. The length of this array is total_cell_nodes. It should be sized as total sum of the number of nodes composing each cell. The input should be arranged as follows. Consider the n^th node of cell i to be c_n^i which is equal to the local index of the corresponding node in the nodes array. Two cells, the first with 5 nodes and the second with 4 would then be defined via this array as: $(c_1^1, c_2^1, c_3^1, c_4^1, c_5^1, c_1^2, c_2^2, c_3^2, c_4^2)$ with the nodes_per_cell array reading $(5, 4)$. The number of nodes per cell is defined by the topology of the cell block given by the associated entry in block_topologies.
- `cell_topologies`: Topologies of the cells. The length of this array is local_num_cells. Give the cell topology type for each cell in the list.

typedef void (***DTK_BoundarySizeFunction**)(void *user_data, size_t *local_num_faces)

Prototype function to get the size parameters for a boundary.

A boundary is a collection of cells (this includes faces of both polyhedrons and cells with fixed topologies) that coincide with a physical geometric boundary and the faces of those cells that are on the boundary. Boundaries may be applied to both cell lists and polyhedron lists.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_BOUNDARY_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_faces`: Number of faces owned by this process that are on the boundary. Boundary data may not exist on all MPI ranks in the map communicator - return a size of zero in the case of no data.

typedef void (***DTK_BoundaryDataFunction**)(void *user_data, LocalOrdinal *boundary_cells, unsigned *cell_faces_on_boundary)

Prototype function to get the data for a boundary.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_BOUNDARY_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `boundary_cells`: Indices of the cells on the boundary. The length of this array is local_num_faces. For every face on the boundary give the local id of the cell to which the face belongs.

This array is of rank-1 and of length equal to the number of faces on the boundary. If the list does not have boundary data on the call MPI rank this array will be empty.

- `cell_faces_on_boundary`: Indices of the faces within a given cell that is on the boundary. The length of this array is local_num_faces. For every face on the boundary give the local id of the face relative to its parent cell. This is the local face id relative to the nodes as defined by the canonical cell topology. This array is of rank-1 and of length equal to the number of faces on the boundary. If the list does not have a boundary this array will be empty.

typedef void (***DTK_AdjacencyListSizeFunction**)(void *user_data, size_t *total_adjacencies)

Prototype function to get the size parameters for building an adjacency list.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_ADJACENCY_LIST_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `total_adjacencies`: Total number of adjacencies in the list.

typedef void (***DTK_AdjacencyListDataFunction**)(void *user_data, GlobalOrdinal *global_cell_ids, GlobalOrdinal *adjacent_global_cell_ids, unsigned *adjacencies_per_cell)

Prototype function to get the data for an adjacency list.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_ADJACENCY_LIST_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_cell_ids`: The global ids of the local cells in the list. The length of this array is total_num_cells in the cell list with which this boundary is associated.
- `adjacent_global_cell_ids`: The global ids of the cells adjacent to the local cells in the list. These may live on another process. The length of this array is total_adjacencies.
- `adjacencies_per_cell`: The number of adjacencies each local cell has. These serve as offsets into the adjacent_global_cell_ids array. The length of this array is total_num_cells in the cell list with which this boundary is associated.

typedef void (***DTK_DOFMapSizeFunction**)(void *user_data, size_t *local_num_dofs, size_t *local_num_objects, unsigned *dofs_per_object)

Prototype function to get the size parameters for a degree-of-freedom id map with a single number of dofs per object (i.e. every object is of the same topology/type).

A degree-of-freedom (dof) map assigns globally-unique indices to objects associated with field data. With such a map, each field value (a degree-of-freedom) can be uniquely identified across the entire DTK MPI communicator, even if it is owned or ghosted on multiple MPI ranks. With this unique identification, it is then possible to compose correct communication plans to transfer the data between arbitrary source and target parallel decompositions. This particular version of the dof map assumes that each object is of the same topology and that each object has the same number of degrees of freedom which may be determined by the topology/type of object and the associated discretization type.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_DOF_MAP_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_dofs`: Number of unique degrees of freedom on this process. Objects may share dofs (for example multiple cells may share a single nodal dof) but each dof should only be included once in this count, regardless of how many objects it is shared by.
- `local_num_objects`: Number of objects on this process that have degrees-of-freedom. This value should correspond to the number of nodes, cells, bounding volumes, or other geometric objects that the user has specified in other inputs describing geometry.
- `dofs_per_objects`: Degrees-of-freedom per object. This is a single value in this case as this map assumes all objects have the same number of dofs.

typedef void (***DTK_DOFMapDataFunction**)(void *user_data, GlobalOrdinal *global_dof_ids, LocalOrdinal *object_dof_ids, char *discretization_type)

Prototype function to get the data for a degree-of-freedom id map with a single number of dofs per object (i.e. every object is of the same topology/type).

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_DOF_MAP_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_dof_ids`: Globally unique ids for DOFs on this process. These may or may not be locally owned but every dof for every object defined on this process must be available in this list. This list is of rank-1 and of length equal to the number of degrees of freedom on the local MPI rank. The length of this array is local_num_dofs.
- `object_dof_ids`: For every object of the given type in the object list give the local dof ids for that object. The local dof ids correspond to the index of the entry in `global_dof_ids` and the number of dofs per object is fixed per the specified cell topology and discretization type. The length of this array is local_num_objects * dofs_per_objects.
- `discretization_type`: Type of discretization.

typedef void (***DTK_MixedTopologyDofMapSizeFunction**)(void *user_data, size_t *local_num_dofs, size_t *local_num_objects, size_t *total_dofs_per_object)

Prototype function to get the size parameters for a degree-of-freedom id map with each object having a potentially different number of dofs (e.g. mixed topology cell lists or polyhedron lists where different objects may have different topologies/types).

A degree-of-freedom (dof) map assigns globally-unique indices to objects associated with field data. With such a map, each field value (a degree-of-freedom) can be uniquely identified across the entire DTK MPI communicator, even if it is owned or ghosted on multiple MPI ranks. With this unique identification, it is then possible to compose correct communication plans to transfer the data between arbitrary source and target parallel decompositions. This particular version of the dof map allows each object to have a different topology and therefore each object can have a different number of degrees of freedom as indicated by the user.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_MIXED_TOPOLOGY_DOF_MAP_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `local_num_dofs`: Number of degrees of freedom on this process. Objects may share degrees of freedom (for example multiple cells may share a single nodal dof) but each degree of freedom should only be included once in this count, regardless of how many objects it is shared by.
- `local_num_objects`: Number of objects on this process. This value should correspond to the number of nodes, cells, bounding volumes, or other geometric objects that the user has specified in other inputs describing geometry.
- `total_dofs_per_objects`: Total degrees of freedom per objects. This is the total sum of the number of dofs on each object.

typedef void (***DTK_MixedTopologyDofMapDataFunction**)(void *user_data, GlobalOrdinal *global_dof_ids, LocalOrdinal *object_dof_ids, unsigned *dofs_per_object, char *discretization_type)

Prototype function to get the data for a multiple object degree-of-freedom id map (e.g. mixed topology cell lists or polyhedron lists).

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_MIXED_TOPOLOGY_DOF_MAP_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Pointer to custom user data.
- `global_dof_ids`: Globally unique ids for DOFs on this process. These may or may not be locally owned but every dof for every object defined on this process must be available in this list. This list is of rank-1 and of length equal to the number of degrees of freedom on the local MPI rank. The length of this array is local_num_dofs.
- `object_dof_ids`: For every object of the given type in the object list give the local dof ids for that object. The local dof ids correspond to the index of the entry in `global_dof_ids`. This array represents unstructured rank-2 data. It should be sized as (total sum of the number of dofs defined on each object) or the total sum of the entries in the dof_per_object array. Consider the n^th dof of object i to be d_n^i which is equal to the local index of the corresponding node in the nodes array. Two objects, the first with 5 dofs and the second with 4 would then be defined via this array as: $(d_1^1, d_2^1, d_3^1, d_4^1, d_5^1, d_1^2, d_2^2, d_3^2, d_4^2)$ with the dofs_per_object array reading $(5, 4)$. The length of this array is total_dofs_per_object.
- `dofs_per_object`: Degrees of freedom per object. For every object, list the number of degress of freedom it contains. The length of this array is local_num_objects.
- `discretization_type`: Type of discretization.

typedef void (***DTK_FieldSizeFunction**)(void *user_data, **const** char *field_name, unsigned *field_dimension, size_t *local_num_dofs)

Prototype function to get the size parameters for a field.

A field represents the actual degrees-of-freedom to be transferred by DTK. In many cases fields are directly associated with degree-of-freedom maps (see above) which describe the unique parallel distribution of the field variables and associate them with with the geometry of the source and target. A field is uniquely identified in the user application by a name. When a DTK transfer operator is applied (see map documentation above) the name of the fields to be transferred are subsequently passed to the source and target implementations of this function - the user should then implement this function to return values corresponding with the input field name.

Field must be of size local_num_dofs in the associated dof_id_map.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_FIELD_SIZE_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field. The user implementation of this function should return sizes associated with this field.
- `field_dimension`: Dimension of the field (i.e. 1 for the pressure, or 3 for the velocity in 3-D)
- `local_num_dofs`: Number of degrees of freedom owned by this process.

typedef void (***DTK_PullFieldDataFunction**)(void *user_data, **const** char *field_name, double *field_dofs)

Prototype function to pull data from the application into a field.

By implementing this function, the user is providing the data from the application in an array format so that it may be transferred by a DTK map. Common use cases use this function to pull data from the source user application for the transfer, however, some map instances may pull data from both the source and target.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_PULL_FIELD_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to pull. The user implementation of this function should return values associated with this field.
- `field_dofs`: Degrees-of-freedom for that field. The length of this array is local_num_dofs * field_dimension. Values are blocked by field dimension. The dof values should directly correlate to the global_dof_ids view in the dof id map. This view is rank-2 and should be dimensioned (degree of freedom, field dimension). The length of the first dimension in this view should be the same as the global_dof_ids view in the dof id map. The second dimension indicates an arbitrary field dimension. This allows for scalars, vectors, and tensors to be assigned as degrees of freedom and transferred simultaneously.

typedef void (***DTK_PushFieldDataFunction**)(void *user_data, **const** char *field_name, **const** double *field_dofs)

Prototype function to push data from a field into the application.

In a transfer operation data is typically pushed to the target user application. By implementing this funciton, the user has access to the data in the target application from the result of a DTK map transfer operation.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_PUSH_FIELD_DATA_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to push. The user implementation of this function should assign values associated with this field.
- `field_dofs`: Degrees-of-freedom for the field. The length of this array is local_num_dofs * field_dimension. Values are blocked by field dimension. The dof values should directly correlate to the global_dof_ids view in the dof id map. This view is rank-2 and should be dimensioned (degree of freedom, field dimension). The length of the first dimension in this view should be the same as the

global_dof_ids view in the dof id map. The second dimension indicates an arbitrary field dimension. This allows for scalars, vectors, and tensors to be assigned as degrees of freedom and transferred simultaneously.

typedef void (***DTK_EvaluateFieldFunction**)(void *user_data, **const** char *field_name, **const** size_t num_points, **const** Coordinate *evaluation_points, **const** LocalOrdinal *object_ids, double *values)

Prototype function to evaluate a field at a given set of points in a given set of objects.

This function gives users the ability to use their own interpolant with a DTK transfer operator. This function provides a set of coordinates at which the function should be evaluated and for each point the local id of the object in which the field is located or to which it is nearest. The user then interpolates the field onto this point and returns the result.

Note Register with a user application using *DTK_setUserFunction()* by passing DTK_EVALUATE_FIELD_FUNCTION as the `type` argument.

Parameters

- `user_data`: Custom user data.
- `field_name`: Name of the field to evaluate. The user implementation of this function should evaluate the field associated with this name.
- `num_points`: The number of points at which to evaluate the field.
- `evaluate_points`: Coordinates of the points at which to evaluate the field. The length of this array is num_points * space_dim. As with other coordinate arrays, these values are blocked by spatial dimension. The spatial dimension of the points is defined by the geometry of the problem (e.g. 3D problems have points with 3 dimensions).
- `objects_ids`: ID of the cell/face with respect of which the coordinates are expressed. The length of this array is num_points.
- `values`: Evaluated field values. The length of this array is num_points * field_dimension. Values are blocked by field dimension.

Enums

enum DTK_Error

DTK error codes.

Values:

DTK_SUCCESS = 0

DTK_INVALID_HANDLE = -1

DTK_UNINITIALIZED = -2

DTK_UNKNOWN = -99

enum DTK_MemorySpace

Memory space (where DTK memory is allocated)

A memory space defines the location in which DTK will allocate memory for the user to access. DTK callback functions allocate memory for reading inputs and writing outputs. Users declare the memory space in which they want their input and output arrays allocated via this enumeration. The following are valid values for the memory space enumeration:

DTK_HOST_SPACE: Memory will be allocated on the host in main CPU memory. Memory allocated in this space is not directly accessible on the GPU. If DTK maps are created for and executed on the GPU or other accelerators this memory will be explicitly copied to and from a memory space accessible by the GPU for computation.

DTK_CUDAUVM_SPACE: Memory will be allocated via CUDA unified virtual memory (UVM) for use with an NVIDIA GPU. This memory is automatically paged between host and device allowing users to access this memory both in standard host functions as well as in CUDA global and device functions. It should be noted that if this memory is accessed on the host in an offloading type of scenario then this memory will be paged between host and device when the user accesses the memory on the host and when DTK access the memory on the GPU and thus may incur a performance cost.

Values:

DTK_HOST_SPACE

DTK_CUDAUVM_SPACE

enum DTK_ExecutionSpace

Execution space (where DTK functions execute)

An execution space defines where DTK mapping computations will occur on a compute node. Interpolation, projection, and other mathematical operations on user fields will execute using the programming model/runtime associated with the execution space parameter. The following are valid values for the execution space enumeration:

DTK_SERIAL: DTK kernels will execute in serial on a single CPU thread.

DTK_OPENMP: DTK kernels will execute in parallel on a number of OpenMP threads specified either via the environment variable OMP_NUM_THREADS or via specification to the Kokkos runtime in initialization via kokkos-threads. If kokkos-specific runtime variables are used to specify thread counts, these should be passed at the time on DTK initialization using *DTK_initializeCmd()*.

DTK_CUDA: DTK kernels will execute in parallel on an NVIDIA GPU using the CUDA runtime. The device on which the kernels will be executed can be specified via the Kokkos runtime in initialization via kokkos-device. If kokkos-specific runtime variables are used to specify devices, these should be passed at the time on DTK initialization using *DTK_initializeCmd()*.

Values:

DTK_SERIAL

DTK_OPENMP

DTK_CUDA

enum DTK_FunctionType

Enumeration passed as the `type` argument to *DTK_setUserFunction()* in order to indicate what callback function is being registered with the user application.

Note Callback functions are passed as pointers to functions that take no arguments and return nothing (`void(*)()`) so the value of the DTK_FunctionType enum is necessary to indicate which user function implementation is being registered with the user application interface.

Values:

DTK_NODE_LIST_SIZE_FUNCTION
See *DTK_NodeListSizeFunction()*

DTK_NODE_LIST_DATA_FUNCTION
See *DTK_NodeListDataFunction()*

DTK_BOUNDING_VOLUME_LIST_SIZE_FUNCTION
See *DTK_BoundingVolumeListSizeFunction()*

DTK_BOUNDING_VOLUME_LIST_DATA_FUNCTION
See *DTK_BoundingVolumeListDataFunction()*

DTK_POLYHEDRON_LIST_SIZE_FUNCTION
See *DTK_PolyhedronListSizeFunction()*

DTK_POLYHEDRON_LIST_DATA_FUNCTION
See *DTK_PolyhedronListDataFunction()*

DTK_CELL_LIST_SIZE_FUNCTION
See *DTK_CellListSizeFunction()*

DTK_CELL_LIST_DATA_FUNCTION
See *DTK_CellListDataFunction()*

DTK_BOUNDARY_SIZE_FUNCTION
See *DTK_BoundarySizeFunction()*

DTK_BOUNDARY_DATA_FUNCTION
See *DTK_BoundaryDataFunction()*

DTK_ADJACENCY_LIST_SIZE_FUNCTION
See *DTK_AdjacencyListSizeFunction()*

DTK_ADJACENCY_LIST_DATA_FUNCTION
See *DTK_AdjacencyListDataFunction()*

DTK_DOF_MAP_SIZE_FUNCTION
See *DTK_DOFMapSizeFunction()*

DTK_DOF_MAP_DATA_FUNCTION
See *DTK_DOFMapDataFunction()*

DTK_MIXED_TOPOLOGY_DOF_MAP_SIZE_FUNCTION
See *DTK_MixedTopologyDofMapSizeFunction()*

DTK_MIXED_TOPOLOGY_DOF_MAP_DATA_FUNCTION
See *DTK_MixedTopologyDofMapDataFunction()*

DTK_FIELD_SIZE_FUNCTION
See *DTK_FieldSizeFunction()*

DTK_PULL_FIELD_DATA_FUNCTION
See *DTK_PullFieldDataFunction()*

DTK_PUSH_FIELD_DATA_FUNCTION
See *DTK_PushFieldDataFunction()*

DTK_EVALUATE_FIELD_FUNCTION
See *DTK_EvaluateFieldFunction()*

Functions

const char ***DTK_version**()
Get the current version of DTK.

Return Returns a string containing the version number for DTK.

const char ***DTK_gitCommitHash**()
Get the current repository hash.

Note If the source code is not under revision control (e.g. downloaded as a tarball), this functions returns an error string indicating that it is not a GIT repository.

Return Returns a string containing the revision number.

void **DTK_initialize**()
Initialize the DTK execution environment without any arguments.

This initializes Kokkos if it has not already been initialized.

void **DTK_initializeCmd**(int **argc*, char ****argv*)
Initialize DTK with command line arguments.

This initializes Kokkos if it has not already been initialized.

This version of initialize() effectively calls `Kokkos::initialize( *argc, *argv )`. Pointers to `argc` and `argv` arguments are passed in order to match MPI_Init's interface.

Parameters

- `argc`: Pointer to the number of argument.
- `argv`: Pointer to the argument vector.

bool **DTK_isInitialized**()
Indicate whether DTK has been initialized.

This function may be used to determine whether DTK has been initialized. DTK must be initialized before any other library function can be used such as user function callback registration and map creation.

void **DTK_finalize**()
Finalize DTK.

This function terminates the DTK execution environment. If DTK initialized Kokkos, this also finalizes Kokkos. However, if Kokkos was initialized before DTK, then this function does NOT finalize Kokkos.

const char ***DTK_error**(int *err*)
Get DTK error message.

All DTK functions set `errno` error code upon completion. If a DTK function fails, the code is nonzero. This function provides a way to get the error message associated with the error code. If the error code is unknown, DTK returns a string stating that.

Return Returns corresponding error string. If the error code is 0 (success), return empty string.

Parameters

- `err`: Error number (typically, errno)

DTK_UserApplicationHandle **DTK_createUserApplication**(*DTK_MemorySpace space*)
Create a DTK handle to a user application in a given memory space.

As many handles may be created as desired with each call to this function giving a new and unique handle. All data for user inputs and outputs accessed through function callbacks registered with a given handle will be allocated in the memory space associated with that handle. A call to this function should be associated with an equivalent call to DTK_destroyUserApplication when the handle's lifetime in the program is complete.

Note User application handles must be valid on all ranks of the communicator over which maps are generated. In other words, this function must be called collectively on every rank in that communicator. In the case

where the user's actual application does not exist on a given MPI rank (and therefore is represented by null data), this function must still be called. User implementations of callback functions for applications where the the actual application does not exist should simply return a size of zero for all application functions.

Return DTK_create returns a handle for the user application. All user inputs and outputs accessed through function callbacks associated with this handle will be allocated in the given memory space.

Parameters

- `space`: Execution space for the callback functions that are to be registered using *DTK_setUserFunction()*.

bool **DTK_isValidUserApplication**(*DTK_UserApplicationHandle handle*)

Indicates whether a DTK handle to a user application is valid.

A handle is valid if it was created by *DTK_createUserApplication()* and has not yet been deleted by *DTK_destroyUserApplication()*.

Return true if the given user application handle is valid; false otherwise.

Parameters

- `handle`: The DTK user application handle to check.

void **DTK_destroyUserApplication**(*DTK_UserApplicationHandle handle*)

Destroy a DTK handle to a user application.

Parameters

- `handle`: User application handle. If this handle has already been destroyed or was not created with a call to *DTK_createUserApplication()* then this function does nothing.

DTK_MapHandle **DTK_createMap**(*DTK_ExecutionSpace space*, MPI_Comm *comm*, *DTK_UserApplicationHandle source*, *DTK_UserApplicationHandle target*, **const** char **options*)

Create a DTK handle to a transfer operator.

An options string is used to select the right map and parameters. This string is defined using a JSON syntax with key-value pairs. For example, specifying a nearest neighbor map would be achieved via:

```
char *options = "{ \"Map Type\": \"Nearest Neighbor\" }";
```

Some maps may have many options. These are separated via commas and may be strings or other types of plain-old-data. For example, consider making a map with an integer-valued option and a double-valued option:

```
char *options = "{ \"Map Type\": \"Name Of Map\", "
                 "\"OptionFooInt\": 3, "
                 "\"OptionBarDouble\": 1.32 }";
```

Return DTK_create returns a handle for the map. This handle must be destroyed with *DTK_destroyMap()* when the lifetime of this map has ended in the program.

Parameters

- `space`: Execution space where the map will execute. Operations on user data for transfer operations will occur in this execution space. If the source or target applications reside in memory spaces that are not compatible with this execution space the data will be copied to and from a compatible memory space as needed.

- `comm`: The MPI communicator over which to build the map. Calls to both *DTK_createMap()* and *DTK_applyMap()* should be considered collective communications over this communicator. Note that this communicator must span all of the MPI ranks on which source and target data must be accessed. For example, if the source and target live on the same MPI communicator and therefore the same set of MPI ranks then that communicator should be the one passed to this function assuming that data for solution transfer will be accessed on all MPI ranks. If the source and target applications live on different MPI communicators composed of entirely different sets of MPI ranks then a new communicator that consists of all of the MPI ranks in both source and target communicators should be created and passed to this function. Cases will also arise in which the source or target application may not exist on some ranks of this communicator (e.g. the previously mentioned case of disjoint source and target communicators). In that case, user implementations of callback functions should just return sizes of zero during calls to allocation functions to indicate to DTK that there is no data from the user application on a given MPI rank.
- `source`: Handle to the source application. This handle must be valid on all ranks in the communicator. Function callback implementations for the source should return zero sizes in allocation functions if the user's source application does not exist on the calling MPI rank.
- `target`: Handle to the target application. Data will be transferred from the source and pushed to this application. This handle must be valid on all ranks in the communicator. Data will be pulled from this application and transferred to the target. Function callback implementations for the target should return zero sizes in allocation functions if the user's target application does not exist on the calling MPI rank.
- `options`: Options string for building the map. The contents of this string specify what type of map to create as well as other parameters specific to constructing that type of map. See above for details on composing this option string using JSON format.

bool **DTK_isValidMap** (*DTK_MapHandle handle*)

Indicates whether a DTK handle to a map is valid.

A handle is valid if it was created by DTK_create() and has not yet been deleted by *DTK_destroyUserApplication()*.

Return true if the given map handle is valid; false otherwise.

Parameters

- `handle`: The DTK map handle to check.

void **DTK_applyMap** (*DTK_MapHandle handle*, **const** char **source_field*, **const** char **target_field*)

Apply the DTK map to the given fields.

This function transfers the data from the source user application to the target user application. The fields transferred by this function are indicated by their given names. In practice, an application could implement their field function callbacks to handle multiple fields, thereby allowing the same map instance to transfer many different fields based on the field name.

Note This function call is a collective over the map's communicator.

Note The source and target user application handles associated with the given map instance must still be valid - they cannot have been destroyed before this function is called.

Parameters

- `handle`: Map handle. This handle must be valid on all calling MPI ranks.

- `source_field`: Name of the field in the source user application. This is the field name passed to either *DTK_PullFieldDataFunction()* or *DTK_EvaluateFieldFunction()* depending on the map type and user implementation of the source user application. DTK will read data from this field.
- `target_field`: Name of the field in the target user application. This is the field name passed to *DTK_PushFieldDataFunction()* in the target user application. DTK will write data to this field.

void **DTK_destroyMap**(*DTK_MapHandle handle*)

Destroy a DTK handle to a map.

Parameters

- `handle`: map handle. If this handle has already been destroyed or was not created with a call to *DTK_createMap()* then this function does nothing.

void **DTK_setUserFunction**(*DTK_UserApplicationHandle handle*, *DTK_FunctionType type*, void (**f*))

void **user_data*Register a function as a callback.

This registers a custom user-implemented function as a callback for DTK to communicate with the user application. The registered function implements the interface specified by the function prototype (see prototypes below) associated with the given function type enumeration. When a function is registered, a user has an opportunity to also assign user data to that instance of the function via a `void*`. When the registered user function is called, the pointer to user data is passed back to the function, allowing the user to use additional data as needed in the implementation or to customize the implementation for specific instances of their application.

Note Use the `user_data` pointer to your advantage when implementing user functions. Anything can be assigned to this pointer: a pointer directly to an instance of the actual user data, special data and functions written specifically for solution transfer, or other auxiliary data structures of the user's construction. Whatever you decide to put here will be passed back to you when the function is called - DTK will not modify this data whatsoever.

Parameters

- `handle`: User application handle. The function implementation will be registered with this particular instance.
- `type`: Type of callback function. The interface of the user function should correspond to the function prototype associated with this enumeration.
- `f`: Pointer to user defined callback function.
- `user_data`: Pointer to the user data that will be passed to the callback function when executing it.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

D