ParPyDTK2 Documentation

Release 1.0.0

Qiao Chen

Sep 07, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Background & Motivation

Multi-physics coupling has become one of the popular topics. People try to put different models together and simulate the behaviors in a coupled fashion. With the popularity of *partitioned approach*, i.e. solve different domains with different solvers and couple the interface conditions as those solvers boundary conditions, a robust and accurate interface solution remapping operator is needed. The solution transfer problem on its own is not an easy task, since it involves the following research aspects:

1. **numerical method**, i.e. consistent, conservative, high-order convergence.
2. **geometry and data structure, i.e. efficient and robust treatments of** *mesh association* of two (potentially more) general surfaces that come from different discretization methods (*FEM*, *FVM*, *FDM*, etc.) thus having different resolutions.
3. **parallel rendezvous & HPC, i.e. handling migrating meshes that have** different parallel partitions.

Data Transfer Kit-2.0 (DTK2) is a package that is developed at the Oak Ridge National Laboratory. DTK2 provides parallel solution transfer services with *meshless* (a.k.a. *mesh-free*) methods, which are relatively easy to implement and computational efficient. Particularly, we are interested in its *modified moving least square*[1] method that is an improvement of traditional MLS fitting in terms of robustness on featured geometries.

Mesh-Oriented datABase (MOAB) is an array-based general purpose mesh library with MPI support. Array-based mesh data structure is more efficient in both computational cost and memory usage compared to traditional pointer-based data structures. MOAB has been adapted in DTK2, so we choose to use it as our mesh database for this work.

In multi-physics coupling, a flexible software framework is must. The fact is: the physics solvers may be implemented in different programming languages or shipped as executable binaries (typically commercial codes), this makes using static languages difficult. Python, on the other hand, can easily glue different languages together and drive executable binaries smoothly. Its built-in reference counting, garbage collection, and pass-by-reference make it as one of the best choices for developing multi-physics coupling frameworks. In addition, MPI is well supported through the mpi4py package. **This motivates us to develop a Python interface for DTK2!**

1.2 License

This package is distributed under MIT License. For detailed information, please take a look at the LICENSE file.

[1] 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).

1.3 About Me

I am a Ph.D. candidate who work with Dr. Jim Jiao on high-order numerical methods. This work is for testing our software framework of multi-physics coupling in general, *conjugate heat transfer* (CHT) in particular.

Note: Please be aware that I may not have time to maintain this package.

CHAPTER

TWO

INSTALLATION

Installing this package is not a trivial task due to its heavy dependencies. ParPyDTK2 has the following installation requirements:

1. C++11 compiler
2. MPI
3. MOAB
4. DTK2 and Trilinos
5. Python >= 3.5
6. mpi4py
7. NumPy
8. setuptools

And both MOAB and DTK2 have their own dependencies.

In addition, to build the documentation, the following packages are needed:

1. Sphinx
2. Doxygen
3. breathe
4. numpydoc

The good news is you can install these easily through `pip`.

2.1 Install MOAB

The MOAB official README has a very clear description of the installation process. Here we take an excerpt from our MOAB Docker image building script:

```
$ git clone --depth=1 https://bitbucket.org/fathomteam/moab.git
$ cd moab
$ autoreconf -fi
$ ./configure \
    --prefix=/usr/local \
    --with-mpi \
    CC=mpicc \
    CXX=mpicxx \
    FC=mpif90 \
```

(continues on next page)

(continued from previous page)

```
    F77=mpif77 \
    --enable-optimize \
    --enable-shared=yes \
    --with-blas=-lopenblas \
    --with-lapack=-lopenblas \
    --with-scotch=/usr/lib \
    --with-metis=/usr/lib/x86_64-linux-gnu \
    --with-eigen3=/usr/include/eigen3 \
    --with-x \
    --with-cgns \
    --with-netcdf \
    --with-hdf5=/usr/lib/hdf5-openmpi \
    --with-hdf5-ldflags="-L/usr/lib/hdf5-openmpi/lib" \
    --enable-ahf=yes \
    --enable-tools=yes
$ make && sudo make install
```

Notice that this is for system installation. Install to your preferred locations if you don't have root access. Also, turn off those optional packages if you don't have them, only MPI and HDF5 are necessary.

Warning: You must build it into a shared object!

Note: If you use Ubuntu >= 17.10, all those optional packages are available through `apt`.

2.2 Install DTK2

DTK2 is shipped as a sub-module of Trilinos, so installing Trilinos is needed. For people who are not familiar with Trilinos, this can be tricky. Therefore, an excerpt from our DTK2 Docker image building script might be helpful:

```
$ export TRILINOS_VERSION=12-12-1
$ git clone --depth 1 --branch trilinos-release-${TRILINOS_VERSION}
$ cd Trilinos
$ git clone --depth 1 --branch dtk-2.0 \
    https://github.com/unifem/DataTransferKit.git
$ mkdir build && cd build
$ cmake \
    -DCMAKE_INSTALL_PREFIX:PATH=/usr/local \
    -DCMAKE_BUILD_TYPE:STRING=RELEASE \
    -DCMAKE_VERBOSE_MAKEFILE:BOOL=OFF \
    -DCMAKE_SHARED_LIBS:BOOL=ON \
    -DTPL_ENABLE_MPI:BOOL=ON \
    -DTPL_ENABLE_Boost:BOOL=ON \
    -DBoost_INCLUDE_DIRS:PATH=/usr/include/boost \
    -DTPL_ENABLE_Libmesh:BOOL=OFF \
    -DTPL_ENABLE_MOAB:BOOL=ON \
    -DMOAB_INCLUDE_DIRS=$MOAB_ROOT/include \
    -DMOAB_LIBRARY_DIRS=$MOAB_ROOT/lib \
    -DTPL_ENABLE_Netcdf:BOOL=ON \
    -DTPL_ENABLE_BinUtils:BOOL=OFF \
    -DTrilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=OFF \
```

(continues on next page)

(continued from previous page)

```
    -DTrilinos_ENABLE_ALL_PACKAGES=OFF \
    -DTrilinos_EXTRA_REPOSITORIES="DataTransferKit" \
    -DTrilinos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
    -DTrilinos_ASSERT_MISSING_PACKAGES:BOOL=OFF \
    -DTrilinos_ENABLE_TESTS:BOOL=OFF \
    -DTrilinos_ENABLE_EXAMPLES:BOOL=OFF \
    -DTrilinos_ENABLE_CXX11:BOOL=ON \
    -DTrilinos_ENABLE_Tpetra:BOOL=ON \
    -DTpetra_INST_INT_UNSIGNED_LONG:BOOL=ON \
    -DTPL_ENABLE_BLAS:BOOL=ON \
    -DTPL_BLAS_LIBRARIES=/usr/lib/x86_64-linux-gnu/libopenblas.so \
    -DTPL_ENABLE_LAPACK:BOOL=ON \
    -DTPL_LAPACK_LIBRARIES=/usr/lib/x86_64-linux-gnu/libopenblas.so \
    -DTPL_ENABLE_Eigen:BOOL=ON \
    -DTPL_Eigen_INCLUDE_DIRS=/usr/include/eigen3 \
    -DTrilinos_ENABLE_DataTransferKit=ON \
    -DDataTransferKit_ENABLE_DBC=ON \
    -DDataTransferKit_ENABLE_TESTS=ON \
    -DDataTransferKit_ENABLE_EXAMPLES=OFF \
    -DDataTransferKit_ENABLE_ClangFormat=OFF \
    -DTPL_ENABLE_BoostLib:BOOL=OFF \
    -DBUILD_SHARED_LIBS:BOOL=ON
$ make && sudo make install
```

Again, this assumes root access, adjust this based on your situation. DTK2 needs to stay in the root directory of Trilinos and be turned on through switches `DTrilinos_EXTRA_REPOSITORIES` and `DTrilinos_ENABLE_DataTransferKit`. The environment var `MOAB_ROOT` is the place where you install MOAB.

Note: We recommend that install DTK2 from our unifem forked repo or my personal forked repo since we may add/modify the source codes to make DTK2 more advanced.

2.3 Install ParPyDTK2

Once you have the dependencies setup, installing ParPyDTK2 can be very easy. The easiest way is through PyPI:

```
$ sudo pip3 install parpydtk2
```

However, this assumes that ParPyDTK2 can find MOAB and DTK2 on the system. With different specifications of `install` command, ParPyDTK2 can automatically add different paths to search for MOAB and DTK2.

```
$ pip3 install parpydtk2 --user
```

will assume MOAB and DTK2 can be found in `USER_BASE/{include,lib}`.

```
$ pip3 install parpydtk2 --prefix=...
```

will allow ParPyDTK2 to search MOAB and DTK2 under the `prefix` directory.

The preferred way is to define the environment variables `PARPYDTK2_MOAB_ROOT` and `PARPYDTK2_DTK_ROOT` before you do `pip install`. For instance,

```
$ export PARPYDTK2_MOAB_ROOT=/path/to/moab/root
$ export PARPYDTK2_DTK_ROOT=/path/to/dtk/root
$ pip3 install parpydtk2 --user
```

Warning: We don't mark mpi4py as installation dependency, so you need to install it manually before you install ParPyDTK2. `pip3 install mpi4py` is just fine.

Of course, you can install from source, which can be obtained here. Just make sure you have all Python *dependencies* installed.

```
$ git clone -b parallel https://github.com/chiao45/parpydtk2.git
$ cd parpydtk2
$ python3 setup.py install --user
```

2.4 Using our Docker container

You can try the package through our pre-built Docker container. Two driver scripts are provided in order to easily use the container:

1. parpydtk2_desktop.py
2. parpydtk2_jupyter.py

The former will launch a desktop environment through VNC, while the latter will run the container as a Jupyter server.

CHAPTER

THREE

SOME DETAILS

3.1 Meshless/Mesh-free

Point clouds are easier to use (for the end user) compared to meshes. However, there are some drawbacks: 1) using meshes can achieve linear time complexity for mesh associations[1]; 2) numerical integrations are not easy; 3) handling cell-averaged data fields, e.g. solutions that come from FVM, potential can lead to problems.

For 3), typically, people assume cell-averaged data to be cell-centered data, which limits to 2nd-order of accuracy. (Notice that this is actually not a big problem in practice.)

3.2 Global IDs/Handles

MOAB uses global IDs for parallel communications as well as DTK2. Therefore, ParPyDTK2 expects the user to provide this information. For most applications, this can be computed offline.

The global IDs are unique handles of the vertices in a point cloud. For instance, if one wants to distribute two triangles, then each of them has local IDs from 0 to 2, but unique global IDs that range from 1 to 4.

Note: ParPyDTK2 uses 1-based indexing for global IDs

3.3 Treatment of Empty Partitions

Both MOAB and DTK2 don't natively support empty partitions, which occur pretty frequently in practice. For instance, couple a serial solver with a parallel solver, or couple a commercial code with an open-sourced one through socket and the incoming data partition graph from the commercial code probably doesn't align with that of the open-sourced side.

To support empty partitions, we duplicate the first node in the master process to the processes that are empty. However, this is not complete yet, because the user is not aware of this, so that the values between across duplicated nodes are not synchronized. Of course, we don't want to let the user to explicitly handle this extra layer of communication. To resolve this, a member function, called `resolve_empty_partitions()`, is added to class `IMeshDB` and must be called **collectively** whenever the user updates the field values through `assign_field()`.

[1] Jiao X, Edelsbrunner X, Heath MT. Mesh association: formulation and algorithms. In 8th International Meshing Roundtable. Sandia Report, South Lake Tahoe, CA, 1999; 75–82.

CHAPTER FOUR

A DEMO

Here, we show a demo for transferring solutions between two meshes of the unit square. We let the `blue` mesh participant run in parallel with two cores, while the `green` side is treated as a serial mesh.

```
import numpy as np
from mpi4py import MPI
from parpydtk2 import *

comm = MPI.COMM_WORLD

blue = IMeshDB(comm)
green = IMeshDB(comm)
assert comm.size == 2
rank = comm.rank
```

For demo purpose, we construct the meshes globally.

```
# create blue meshes on all processes
cob = np.empty(shape=(16, 3), dtype=float, order='C')
dh = 0.3333333333333333
index = 0
x = 0.0
for i in range(4):
    y = 0.0
    for j in range(4):
        cob[index] = np.asarray([x, y, 0.0])
        index += 1
        y += dh
    x += dh

# global IDs, one based
bgids = np.arange(16, dtype='int32')
bgids += 1
```

The `blue` side has 16 nodes, the following is for the `green` side:

```
# create green meshes on all processes
cog = np.empty(shape=(36, 3), dtype=float, order='C')
dh = 0.2
index = 0
x = 0.0
for i in range(6):
    y = 0.0
    for j in range(6):
        cog[index] = np.asarray([x, y, 0.0])
```

(continues on next page)

(continued from previous page)

```
        index += 1
        y += dh
    x += dh

# global IDs
ggids = np.arange(36, dtype='int32')
ggids += 1
```

The `green` participant has 36 nodes. The next step is to put the data in the two mesh databases.

```
46 # creating the mesh database
47 blue.begin_create()
48 # local vertices and global IDs
49 lcob = cob[8 * rank:8 * (rank + 1), :].copy()
50 lbgids = bgids[8 * rank:8 * (rank + 1)].copy()
51 blue.create_vertices(lcob)
52 blue.assign_gids(lbgids)
53 blue.create_field('b')
54
55 # do not use trivial global ID strategy
56 blue.finish_create(False)
```

As we can see, we equally distributed the mesh into the two cores as well as the corresponding global IDs. In line 56, the `False` flag indicates that the mesh database should use the user-provided global IDs.

> **Warning:** Creating vertices and fields, and assigning global IDs must be called within *`begin_create()`* and *`finish_create()`*! Otherwise, exceptions are thrown.

Here is the treatment for the "serial" participant:

```
# NOTE that green is assumed to be serial mesh
green.begin_create()
# only create on master rank
if not rank:
    green.create_vertices(cog)
green.create_field('g')
# since green is serial, we just use the trivial global IDs
green.finish_create()  # empty partition is resolve here

assert green.has_empty()
```

As we can see, only the master process has data.

> **Note:** The *duplicated* node is handled inside *`finish_create()`*

With the two participants ready, we can now create our *`Mapper`*.

```
# create our analytical model, i.e. 10+sin(x)*cos(y)
bf = 10.0 + np.sin(lcob[:, 0]) * np.cos(lcob[:, 1])
gf = np.sin(cog[:, 0]) * np.cos(cog[:, 1]) + 10.0

# Construct our mapper
mapper = Mapper(blue=blue, green=green)
```

(continues on next page)

(continued from previous page)

```
75
76  # using mmls, blue radius 1.0 green radius 0.6
77  mapper.method = MMLS
78  mapper.radius_b = 1.0
79  mapper.radius_g = 0.6
80  mapper.dimension = 2
81
82  # Mapper initialization region
83  mapper.begin_initialization()
84  mapper.register_coupling_fields(bf='b', gf='g', direct=B2G)
85  mapper.register_coupling_fields(bf='b', gf='g', direct=G2B)
86  mapper.end_initialization()
```

Line 69-70 just create an analytic model for error analysis. Line 77-80 are for parameters, for this case, we use MMLS (default) with `blue` side radius 1.0 and `green` side radius 0.6 for searching.

The important part is from line 83 to 86. Particularly speaking, the function *`register_coupling_fields()`*. It takes three parameters, where the first two are string tokens that represents the data fields in `blue` and `green`. The `direct` is to indicate the transfer direction, e.g. *`B2G`* stands for blue to green.

> **Warning:** *`register_coupling_fields()`* must be called within *`begin_initialization()`* and *`end_initialization()`*.

```
88  # NOTE that the following only runs on green master mesh
89  if not rank:
90      green.assign_field('g', gf)
91  # Since green is serial and has empty partition, we must call this to
92  # resolve asynchronous values
93  green.resolve_empty_partitions('g')
```

The above section is to assign values on the `green` participant. Notice that it is a "serial" mesh, so we only assign values on the master process. But resolve the *duplicated* node is needed, this is done in line 93.

Finally, the solution transfer part is pretty straightforward:

```
94  # solution transfer region
95  mapper.begin_transfer()
96  mapper.transfer_data(bf='b', gf='g', direct=G2B)
97  err_b = (bf - blue.extract_field('b'))/10
98  mapper.transfer_data(bf='b', gf='g', direct=B2G)
99  err_g = (gf - green.extract_field('g'))/10
100 mapper.end_transfer()
101
102 comm.barrier()
103
104 print(rank, 'blue L2-error=%.3e' % (np.linalg.norm(err_b)/np.sqrt(err_b.size)))
105 if rank == 0:
106     print(0, 'green L2-error=%.3e' % (np.linalg.norm(err_g)/np.sqrt(err_g.size)))
```

This code can be obtained here parallel2serial.py.

CHAPTER

FIVE

API

5.1 `ParPyDTK2` Python API

5.1.1 Default

Main module interface of ParPyDTK2

`parpydtk2.`**`B2G`**
: *bool* – boolean flag of `True` denotes transferring direction from `blue` to `green`

`parpydtk2.`**`G2B`**
: *bool* – boolean flag of `False` denotes transferring direction from `green` to `blue`

`parpydtk2.`**`MMLS`**
: *int* – flag (0) represents using *modified moving least square* method

`parpydtk2.`**`SPLINE`**
: *int* – flag (1) represents using *spline interpolation* method

`parpydtk2.`**`N2N`**
: *int* – flag (2) represents using *nearest node projection* method

`parpydtk2.`**`WENDLAND2`**
: *int* – flag (0) represents using *Wendland 2nd-order* RBF weights

`parpydtk2.`**`WENDLAND4`**
: *int* – flag (1) represents using *Wendland 4th-order* RBF weights

`parpydtk2.`**`WENDLAND6`**
: *int* – flag (2) represents using *Wendland 6th-order* RBF weights

`parpydtk2.`**`WU2`**
: *int* – flag (3) represents using *Wu 2nd-order* RBF weights

`parpydtk2.`**`WU4`**
: *int* – flag (4) represents using *Wu 4th-order* RBF weights

`parpydtk2.`**`WU6`**
: *int* – flag (5) represents using *Wu 6th-order* RBF weights

`parpydtk2.`**`BUHMANN3`**
: *int* – flag (6) represents using *Buhmann 3rd-order* RBF weights

`parpydtk2.`**`get_include`**`()`
: Get the abs include path

5.1.2 Error Handling

The error handler module

parpydtk2.error_handle.**ERROR_CODE**

int – set this to nonzero values one exceptions have been raised

Examples

```
>>> import parpydtk2 as dtk
>>> try:
...     # your programs here
... except Expection:
...     dtk.error.ERROR_CODE = 1
...     raise
```

5.1.3 Interface Mesh & Mapper

class parpydtk2.**IMeshDB**(*comm=None*)

Interface mesh database

ParPyDTK2 utilizes MOAB as the underlying mesh database. MOAB is an array based mesh library that is adapted by DTK2. With array based mesh library, the memory usage and computational cost are lower than typical pointer based data structure. The mesh concept in this work is simple since only meshless methods are ultilized, the only additional attribute one needs is the *global IDs/handles*, which are used by both MOAB and DTK2. For most applications, the global IDs can be computed offline.

One thing is not directly supported by IMeshDB is I/O. However, since this is a Python module and only points clouds are needed, one can easily uses a tool (e.g. meshio) to load the mesh.

comm

MPI.Comm – MPI communicator

ranks

int – size of comm

rank

int – rank of comm

size

int – point cloud size, i.e. number of vertices

bbox

np.ndarray – local bounding box array of shape (2,3)

gbbox

np.ndarray – global bounding box array of shape (2,3)

Constructor

Parameters **comm** (*MPI.Comm (optional)*) – if no communicator or None is passed in, then MPI_COMM_WORLD will be used

Examples

```
>>> # implicit communciator
>>> import parpydtk2 as dtk
>>> mdb = dtk.IMeshDB()
```

```
>>> # explicit communciator
>>> from mpi4py import MPI
>>> import parpydtk2 as dtk
>>> mdb = dtk.IMeshDB(MPI.COMM_WOLRD)
```

assign_field(*self*, *unicode field_name*, *ndarray values*)
Assign values to a field

Note: *values* size must be at least size*dim

Parameters

- **field_name** (*str*) – name of the field
- **values** (*np.ndarray*) – input source values

See also:

extract_field() extract value from a field

size check the size of a mesh set

assign_gids(*self*, *__Pyx_memviewslice gids*)
Assign global IDs

Internally, both DTK and MOAB use so-called global IDs/handles communications. Each node has its own local IDs/handles and a unique global ID.

Parameters **gids** (*np.ndarray*) – global IDs

See also:

create_vertices() create vertices

extract_gids() extract global IDs

bbox
np.ndarray – local bounding box

The bounding box is stored simply in a 2x3 array, where the first row stores the maximum bounds while minimum bounds for the second row.

Warning: Bounding box is valid only after *finish_create()*.

See also:

gbbox global bounding box

begin_create (*self*)
Begin to create/manupilate the mesh

This function must be called in order to let the mesh databse be aware that you will create meshes.

See also:

finish_create() finish creating mesh

comm
MPI.Comm – communicator

create_field (*self*, *unicode field_name*, *int dim=1*)
Create a data field for solution transfer

This is the core function to register a field so that you can then transfer its values to other domains. The `dim` parameter determines the data type of the field. By default, it's 1, i.e. scalar fields. For each node, a tensor of (1x``dim``) can be registered. For instance, to transfer forces and displacements in FSI applications, `dim` is 3 (for 3D problems).

Parameters

- **field_name** (*str*) – name of the field
- **dim** (*int*) – dimension of the field, i.e. scalar, vector, tensor

Examples

```
>>> from parpydtk2 import *
>>> mdb1 = IMeshDB()
>>> mdb1.begin_create()
>>> mdb1.create_field('heat flux')
```

create_vertices (*self*, *__Pyx_memviewslice coords*)
Create a set of coordinates

Note: The `coords` must be C-ordering with ndim=2!

Parameters **coords** (*np.ndarray*) – nx3 coordinates in double precision

See also:

assign_gids() assign global IDs

extract_vertices() extract vertex coordinates

Examples

```
>>> from parpydtk2 import *
>>> import numpy as np
>>> mdb1 = IMeshDB()
>>> mdb1.begin_create()
>>> verts = np.zeros((2,3))  # two nodes
```

(continues on next page)

(continued from previous page)

```
>>> verts[1][0] = 1.0
>>> mdb1.create_vertices(verts)
```

empty (*self*)
Check if this is an empty partition

extract_field (*self*, *unicode field_name*, *__Pyx_memviewslice buffer=None*, *reshape=False*)
Extact the values from a field

> **Warning:** if *buffer* is passed in, it must be 1D

Parameters

- **field_name** (*str*) – name of the field
- **buffer** (*np.ndarray*) – 1D buffer
- **reshape** (*bool*) – *True* if we reshape the output, only for vectors/tensors

Returns field data values

Return type np.ndarray

extract_gids (*self*)
Extract global IDs/handles

> **Warning:** This function should be called once you have finished `assign_gids()`.

Returns array of `size` that stores the integer IDs

Return type np.ndarray

extract_vertices (*self*)
Extract coordinate

> **Warning:** This function should be called once you have finished `create_vertices()`.

Returns (nx3) array that stores the coordinate values

Return type np.ndarray

See also:

`size` get the mesh size

field_dim (*self*, *unicode field_name*)
Check the field data dimension

Parameters **field_name** (*str*) – name of the field

Returns data field dimension of *field_name*

Return type int

finish_create (*self*, *trivial_gid=True*)
finish mesh creation

This method finalizes the interface mesh database by communicating the bounding boxes and empty partitions. Also, setting up the DTK managers happens here.

> **Warning:** You must call this function once you have done with manupilating the mesh, i.e. vertices and global IDs.

Parameters **trivial_gid** (`bool`) – *True* if we use MOAB trivial global ID computation

Notes

By `trivial_gid`, it means simply assigning the global IDs based on the size of the mesh. This is useful in serial settings or transferring solutions from a serial solver to a partitioned one.

gbbox
np.ndarray – global bounding box

The bounding box is stored simply in a 2x3 array, where the first row stores the maximum bounds while minimum bounds for the second row.

> **Warning:** Bounding box is valid only after `finish_create()`.

See also:

bbox local bounding box

has_empty (*self*)
Check if an empty partition exists

has_field (*self*, *unicode field_name*)
Check if a field exists

Parameters **field_name** (`str`) – name of the field

Returns *True* if this meshdb has *field_name*

Return type bool

rank
int – get the rank

ranks
int – Get the communicator size

resolve_empty_partitions (*self*, *unicode field_name*)
Resolve asynchronous values on empty partitions

ParPyDTK2 doesn't expect `assign_field()` should be called collectively. Therefore, a collective call must be made for resolving empty partitions.

> **Warning:** This must be called collectively even on empty partitions

Note: You should call this function following assignment

Parameters **`field_name`** (*`str`*) – name of the field

Examples

```
>>> if rank == 0:
...     mdb.assign_field('flux', values)
>>> mdb.resolve_empty_partitions('flux')
```

Notes

This API is not available in C++ level, therefore, one needs to implement this if he/she wants to use the C++ API.

`size`
int – Get the size of a set

class `parpydtk2.`**`Mapper`**(*blue*, *green*, *profiling=True*)
DTK2 wrapper

The meshless methods in DTK2, including *modified moving least square*, *spline interpolation* and *nearest node projection* methods are wrapped within this class. The most advanced method is the MMLS fitting, which is my personal recommendataion.

`blue_mesh`
`IMeshDB` – blue mesh participant

`green_mesh`
`IMeshDB` – green mesh participant

`comm`
MPI.Comm – MPI communicator

`ranks`
int – size of comm

`rank`
int – rank of comm

`dimension`
int – spatial dimension

`method`
int – method flag, either MMLS, SPLINE, or N2N

`basis`
int – flag of basis function for weighting schemes used by MMLS and SPLINE

`knn_b`
int – k-nearest neighborhood for searching on blue_mesh

`knn_g`
int – k-nearest neighborhood for searching on green_mesh

radius_b
float – radius used for searching on blue_mesh

radius_g
float – radius used for searching on green_mesh

Constructor

Parameters

- **blue_mesh** (*IMeshDB*) – blue mesh participant
- **green_mesh** (*IMeshDB*) – green mesh participant
- **profiling** (*bool (optional)*) – whether or not do timing report, default is `True`.

Examples

```
>>> from parpydtk2 import *
>>> blue, green = IMeshDB(), IMeshDB()
>>> # initialize blue and green
>>> mapper = Mapper(blue=blue,green=green)
>>> # do work with mapper
```

basis
int – Get the basis function flag

See also:

method get the method tag

begin_initialization(*self*)
Initialization starter

This is a must-call function in order to indicate mapper that you are about to initialize/register coupling fields

See also:

register_coupling_fields() register coupled fields

end_initialization() finish initialization

begin_transfer(*self*)
Transfer starter

This is a must-call function to inidate the beginning of a transferring block

See also:

end_transfer() transfer closer

tranfer_data() transfer a coupled data fields

blue_mesh
IMeshDB – blue mesh

comm
MPI.Comm – Get the communicator

See also:

`ranks` get the total communicator size

`rank` "my" rank

`dimension`
int – Get the problem dimension

Note: this is the spacial dimension

`end_initialization`(*self*)
Initialization closer

This is a must-call function in order to tell the mapper we are ready

See also:

`begin_initialization()` initialization starter

`end_transfer`(*self*)
Transfer closer

This is a must-call function to indicate we have finished a sequence of transferring requests

See also:

`begin_transfer()` transfer starter

`green_mesh`
`IMeshDB` – green mesh

`has_coupling_fields`(*self*, *unicode bf*, *unicode gf*, *bool direct*)
Check if a coupled fields exists

Returns `True` if (bf,gf) exists in the `direct` direction

Return type bool

`knn_b`
int – KNN of blue mesh

Note: if blue does not use KNN, then -1 returned

See also:

`knn_g` green knn

`knn_g`
int – KNN of green mesh

Note: if green does not use KNN, then -1 returned

See also:

`knn_b` blue knn

method

int – Get the method tag

See also:

basis the basis function and order attribute

radius_b

float – physical domain radius support for blue mesh

Note: if blue does not use RBF-search, then -1.0 returned

See also:

radius_g green radius

radius_g

float – physical domain radius support for green mesh

Note: if green does not use RBF-search, then -1.0 returned

See also:

radius_b blue radius

rank

int – Check "my" rank

See also:

ranks get the total communicator size

comm MPI communicator

ranks

int – Check the total process number

See also:

rank "my" rank

register_coupling_fields(*self*, *unicode bf*, *unicode gf*, *bool direct*)

register a coupled fields

Note: we use boolean to indicate direction

Parameters

- **bf** (*str*) – blue mesh field name
- **gf** (*str*) – green mesh field name
- **direct** (*bool*) – *True* for blue->green, *False* for the opposite

See also:

[`transfer_data()`] transfer a coupled data fields

`set_matching_flag_n2n`(*self, bool matching*)
Set the matching flag for N2N

Note: this function will not throw even if you dont use n2n

Parameters **`matching`** (*`bool`*) – *True* if the interfaces are matching

`transfer_data`(*self, unicode bf, unicode gf, bool direct*)
Transfer (bf, gf) in the `direct` direction

Parameters

- **`bf`** (*`str`*) – blue mesh field name
- **`gf`** (*`str`*) – green mesh field name
- **`direct`** (*`bool`*) – `True` for blue->green, `False` for the opposite

See also:

[`register_coupling_fields()`] register coupled fields

5.2 `ParPyDTK2` C++ API

5.2.1 Common Definitions

group **`common`**

Defines

`handle_moab_error`(__ret)
macro to handle moab error

`throw_error`(__msg)
throw runtime_error exception

`throw_error_if`(__cond, __msg)
conditionally error throw

`throw_noimpl`(__what)
throw not implemented feature error

`throw_noimpl_if`(__cond, __what)
throw not implemented feature error with condition

`show_warning`(__msg)
log warning message in stderr

`show_warning_if`(__cond, __msg)
log warning with condition

show_experimental (__msg)
log experimental warning in stderr

show_experimental_if (__cond, __msg)
log experimental warning in stderr with condition

show_info (__msg, __rank)
show information in parallel

show_info_master (__msg, __rank)
show information only on master rank

streamer (__rank)
streaming message with specific rank

streamer_master (__rank)
streaming messages only on the master process

Typedefs

typedef entity_t
MOAB entity handle.

Enums

enum [anonymous]
root set

Values:

root_set = 0

5.2.2 Field Variables

class FieldData
a representation of MOAB tag for field data

Public Functions

FieldData (moab::Core &*mdb*, **const** std::string &*field_name*, int *dim* = 1)
constructor with moab instance

Note *set* is not the same as mesh set in MOAB

Parameters

- `mdb`: moab instance
- `field_name`: field name
- `dim`: field dimension

void **assign** (**const** moab::Range &*range*, **const** double **values*)
assign values

Parameters

- `range`: entity ranges, for this work, it should be vertices
- `values`: data values, for vector/tensor, C order is expected

void **assign_1st**(**const** moab::Range &*range*, **const** double **values*)
assign to first node

This function is used by Python for handling empty partitions

Parameters

- `range`: entity ranges
- `values`: values for the first node, at least size of dim

void **extract**(**const** moab::Range &*range*, double **values*) **const**
extract values

Parameters

- `range`: entity ranges, for this work, it should be vertices
- `values`: data values, for vector/tensor, C order is expected

void **extract_1st**(**const** moab::Range &*range*, double **values*) **const**
extract the value from the first node

Parameters

- `range`: entity ranges
- `values`: values for the first node, at least size of dim

operator const std::string&() **const**
brief implicitly cast to string

int **dim**() **const**
check the dimension

int **set**() **const**
check the set ID

const moab::Tag &**tag**() **const**
get MOAB tag

Protected Attributes

moab::Core &**mdb_**
reference to moab instance

std::string **fn_**
field name

int **set_**
set count

int **dim_**
: field dimension

moab::Tag **tag_**
: moab tag

class FieldDataSet
: a set of field data

Public Types

typedef *base_t*::iterator **iterator**
: interator type

typedef *base_t*::const_iterator **const_iterator**
: constant iterator

Public Functions

virtual ~FieldDataSet()
: destructor

bool **has_field**(**const** std::string &*fn*) **const**
: check if a field exist

Parameters

- `fn`: field name

void **create**(moab::Core &*mdb*, **const** std::string &*field_name*, int *dim* = 1)
: create an data field

Note *set* is not the same as mesh set in MOAB

Parameters

- `mdb`: moab data base
- `field_name`: field name
- `dim`: field dimension

FieldData &**operator[]**(**const** std::string &*fn*)
: get a reference to a field data

Note this overloads the base operator[]

Parameters

- `fn`: field name

const *FieldData* &**operator[]**(**const** std::string &*fn*) **const**
: get a const reference to a field data

Parameters

- `fn`: field name

iterator **begin**()
    get the first iterator

iterator **end**()
    get the end iterator

const_iterator **begin**() **const**
    get the constant iterator

const_iterator **end**() **const**
    get the constant end iterator

const_iterator **cbegin**() **const**
    get the constant iterator

const_iterator **cend**() **const**
    get the constant end iterator

Protected Attributes

base_t **fs_**
    fields

Private Types

typedef std::unordered_map<std::string, *FieldData* *> **base_t**
    data structure

5.2.3 Interface Mesh Database

class IMeshDB
    interface mesh database, build on top of MOAB

imesh_py_interface

void **begin_create**()
    begin to create mesh

void **create_vset**()
    create a new vertex set

void **create_vertices**(int *nv*, **const** double **coords*)
    create vertices

Parameters

- `nv`: number of vertices
- `coords`: coords values

void **extract_vertices**(double **coords*) **const**
    extract assigned coordinates

Note coords must be at least n*3 where n is the size of the mesh

Parameters

- `coords`: coordinates

void **assign_gids** (int *nv*, **const** int **gids*)
assign global IDs

Note *gids* should be one-based indices

Parameters

- `nv`: number of local vertices
- `gids`: global IDs

void **extract_gids** (int **gids*) **const**
extract global IDs

Parameters

- `gids`: global IDs

void **finish_create** (bool *trivial_gid* = true)
finish manupilating the mesh

NOTE that if your input mesh is element-based partition and the vertices you create are mesh nodes, then you have to specify the correct global IDs in parallel. However, if the coordinates are face centres, then the global IDs can be trivially computed by MOAB since there are no shared entities cross different processes.

Parameters

- `trivial_gid`: *true* if we let MOAB to compute the GID

bool **empty** () **const**
check if empty partition

bool **has_empty** () **const**
check if any of the process has an empty partition

const std::vector<int> &**_m2s** () **const**
get a reference to the m2s pattern

Note This is used in Python level as "private" thus having "_"

int **size** () **const**
check mesh size

void **get_bbox** (double **v*) **const**
get bounding box

Parameters

- `v`: values

void **get_gbbox** (double **v*) **const**
get global bounding box

Parameters

- `v`: values

void **`create_field`**(**`const`** std::string &*field_name*, int *dim* = 1)
create a field

Parameters

- `field_name`: field name
- `dim`: field dimension

bool **`has_field`**(**`const`** std::string &*field_name*) **`const`**
check if we have a field

Parameters

- `field_name`: field name

int **`field_dim`**(**`const`** std::string &*field_name*) **`const`**
check field dimension

Parameters

- `field_name`: field name

void **`assign_field`**(**`const`** std::string &*field_name*, **`const`** double **values*)
assign a value to a field

Parameters

- `field_name`: field name
- `values`: field data values

void **`_assign_1st`**(**`const`** std::string &*field_name*, **`const`** double **values*)
assign to the first node

This function is used by Python to resolve the issues when empty empty partitions happen. Therefore, this function has an "_" prefix to indicate "private" usage!

Parameters

- `field_name`: field name
- `values`: field data values, at least size of field dimension

void **`extract_field`**(**`const`** std::string &*field_name*, double **values*) **`const`**
extract value

Parameters

- `field_name`: field name
- `values`: field data values

void **_extract_1st** (**const** std::string &*field_name*, double **values*) **const**
extract first value

This function is used by Python to resolve the issues when empty empty partitions happen. Therefore, this function has an "_" prefix to indicate "private" usage!

Parameters

- `field_name`: field name
- `values`: field data values

Public Functions

IMeshDB (MPI_Comm *comm* = MPI_COMM_WORLD)
constructor with communicator

Parameters

- `comm`: communicator

int **ranks** () **const**
get total ranks

int **rank** () **const**
get my rank

MPI_Comm **comm** () **const**
get the communicator

Teuchos::RCP<moab::ParallelComm> **pcomm** () **const**
get mesh

std::vector<DataTransferKit::MoabManager> &**mangers** ()
get the manger

void **set_dimension** (int *dim*)
set geometry dimension

Parameters

- `dim`: dimension

bool **ready** () **const**
check if ready

dtk_field_t &**dtk_fields** ()
get the dtk fields

Protected Types

typedef std::unordered_map<std::string, std::pair<int, Teuchos::RCP<Tpetra::MultiVector<double, int, DataTransferKit::Sup
handy typedef

Protected Attributes

moab::Core **mdb_**
moab instance

Teuchos::RCP<moab::ParallelComm> **par_**
moab parallel interface

std::vector<*entity_t*> **vsets_**
vertex sets

std::vector<moab::Range> **locals_**
local vertex range

std::vector<std::array<double, 6>> **bboxes_**
bounding boxes

std::vector<std::array<double, 6>> **gbboxes_**
global bounding boxes

FieldDataSet **fields_**
field data set

bool **created_**
flag to indicate whether users are done with creating mesh

moab::Tag **gidtag_**
global ID tag

moab::Tag **parttag_**
partition tag

bool **usergid_**
flag to indicate if we have user computed global ID

std::vector<DataTransferKit::MoabManager> **mngrs_**
DTK MOAB manager.

dtk_field_t **dtkfields_**
DTK multi vector for MOAB tags.

bool **empty_**
check empty partition

bool **has_empty_**
check if any process is empty

std::vector<int> **m2s_**
comm pattern for master2slaves for handling empty partitions

Private Functions

void **init_**(bool *del* = false)
helper for clean up mesh

Parameters

- `del`: whether or not delete mesh

void **reset_vecs_**()
handle all vectors

void **init_bbox_**(int *i* = -1)
initialize empty bounding boxes

Parameters

- `i`: index if < 0 then init all

void **cmpt_bboxes_**()
compute all bounding box

5.2.4 Solution Mapper

class Mapper
the mapper interface for interface solution transfer

mapper_py_interface

int **ranks**() **const**
get the ranks

int **rank**() **const**
get my rank

MPI_Comm **comm**() **const**
get the communicator

void **set_dimension**(int *dim*)
set dimension

Parameters

- `dim`: geometry dimension

void **use_mmls**()
use moving least square, this is the default method

See *use_spline*, *use_n2n*

void **use_spline**()
use spline interpolation method

See *use_mmls*, *use_n2n*

void **use_n2n**(bool *matching* = false)
use node 2 node project

See *use_mmls*, *use_spline*

Parameters

- `matching`: are the interfaces mathing?

void **set_basis** (int *basis*)
set basis function, default is Wendland 4th order

See BasisFunctions

Parameters

- `basis`: basis function and order

void **use_knn_b** (int *knn*)
use knn for blue mesh

Parameters

- `knn`: number of nearest neighbors

void **use_knn_g** (int *knn*)
use knn for green mesh

Parameters

- `knn`: number of nearest neighbors

void **use_radius_b** (double *r*)
use radius for blue

See use_knn

Parameters

- `r`: physical domain radius support

void **use_radius_g** (double *r*)
use radius for green

See use_knn

Parameters

- `r`: physical domain radius support

int **check_method** () **const**
check method

int **check_basis** () **const**
check basis

int **knn_b** () **const**
check blue knn

Note if blue does not use knn, then negative value returned

int **knn_g** () **const**
check green knn

double **radius_b** () **const**
check blue radius

Note if blue does not use radius, then -1.0 is returned

double **radius_g**() **const**
check green radius

int **dimension**() **const**
get the dimension

std::shared_ptr<*IMeshDB*> **blue_mesh**() **const**
get blue mesh

std::shared_ptr<*IMeshDB*> **green_mesh**() **const**
get green mesh

void **begin_initialization**()
begin initialization

void **register_coupling_fields**(**const** std::string &*bf*, **const** std::string &*gf*, bool *direct*)
register coupling fields

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b

bool **has_coupling_fields**(**const** std::string &*bf*, **const** std::string &*gf*, bool *direct*)
check if a coupling data fields exists

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b

void **end_initialization**()
end initialization

void **begin_transfer**()
begin to transfer data

void **transfer_data**(**const** std::string &*bf*, **const** std::string &*gf*, bool *direct*)
transfer data

Parameters

- `bf`: blue meshdb field data
- `gf`: green meshdb field data
- `direct`: *true* for b->g, *false* for g->b

void **end_transfer**()
end transfer

Public Functions

Mapper (std::shared_ptr<*IMeshDB*> *B*, std::shared_ptr<*IMeshDB*> *G*, **const** std::string &*version* = "", **const** std::string &*date* = "", bool *profiling* = true)
constructor

Parameters

- `B`: input blue mesh
- `G`: input green mesh
- `version`: passed from Python inteface
- `date`: passed from Python interface
- `profiling`: whether do simple profiling, i.e. wtime

Protected Attributes

std::shared_ptr<*IMeshDB*> **B_**
blue mesh

std::shared_ptr<*IMeshDB*> **G_**
green mesh

int **dim_**
dimension

bool **ready_**
flag to indicate the mapper is ready for transfering

bool **profiling_**
whether do simple profiling

double **timer_**
a simple timer buffer

std::unique_ptr<Teuchos::ParameterList> **opts_**[2]
parameter list

std::map<std::pair<std::string, std::string>, Teuchos::RCP<DataTransferKit::MapOperator>> **operators_**[2]
transfer operators

Protected Static Attributes

DataTransferKit::MapOperatorFactory **factory_**
map factory

Private Functions

void **init_parlist_**()
initialize parameter list

template <bool *_Dir*, **typename** _V>
void **set_search** (**const** std::string &*type*, **const** std::string &*value*, **const** _V &*v*)
helper for set local search

Template Parameters

- `_Dir`: direction
- `_V`: value type

Parameters

- `type`: search type
- `value`: tag for value
- `v`: actual value

template <bool *_Dir*, **typename** _V>
_V **get_search**(**const** std::string &*type*, **const** std::string &*value*, **const** _V &*dft*) **const**
helper for get search info

Template Parameters

- `_Dir`: direction
- `_V`: value type

Parameters

- `type`: search type
- `value`: tag for value
- `dft`: default value

Private Static Functions

static std::string **parse_list_**(Teuchos::ParameterList &*list*)
parse and formatting a parameter list

Parameters

- `list`: parameter list

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

R

S

T

W