# Data Transfer Kit Summary

Stuart R. Slattery
Engineering Physics Department
University of Wisconsin - Madison

January 15, 2013

# Outline

- Development Hisotry
- Domain Model and Geometric Rendezvous
- DTK Algorithms
- Code Example

# What is DTK?

- Collection of geometry-based data mapping algorithms for shared domain probelems
- Data maps allow for efficient movement of data in parallel (e.g. between meshes of a different parallel decomposition)
- Ideally maps are generated at a desireable time complexity (logarithmic)
- Input mesh and geometry data drive the map generation
- Should be viewed as a service providing suite of concrete algorithm implementations

# What DTK Doesn't Do

- Does not provide a general interface for all physics codes to couple to all other physics codes
- Does not provide discretization services (e.g. basis functions)
- Does not provide algorithm implementations for interface-based data transfer
- Does not allocate or deallocate memory in user code

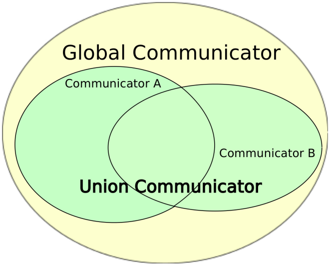
# Software Overview

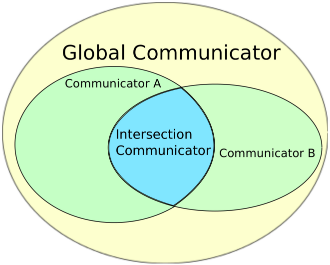
- Preliminary development of mesh-based capabilities during summer 2012 CASL internship at ORNL
- Additional development of geometry-based capabilities during fall 2012
- Implemented in C++
- Heavy use of the Trilinos scientific computing libraries
- Continuous and nightly testing as part of the CASL CDash system
- Open-source BSD 3-clause license
- https://github.com/CNERG/DataTransferKit

# Concepts and Geometric Rendezvous

# Communicators

- DTK handles source and target communicators of arbitrary relation
- Any amount of overlap or lack thereof supported
- A global communicator required (doesn't have to be MPI_COMM_WORLD)



# Shared Domain Problems

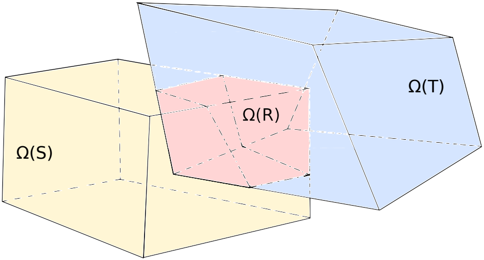



Figure: *Shared domain example.*

*$\Omega(S)$ (yellow) is the source geometry, $\Omega(T)$ (blue) is the target geometry, and $\Omega(R)$ (red) is the shared domain.*

- Defined over a communicator that encapsulates the union of the source and target communicators
- Source and target must be of same geometric dimension
- The rendezvous algorithm leveraged to provide parallel topology maps for shared domains

# Parallel Topology Maps

- An operator, $\mathbf{M}$, that defines the translation of a field, $\mathbf{F}(s)$, from a source spatial domain, $\Omega_S$, to a field, $\mathbf{G}(t)$, in the target spatial domain $\Omega_T$, such that $\mathbf{G}(t) \leftarrow \mathbf{M}(\mathbf{F}(s))$ and $\mathbf{M} : \mathbb{R}^D \rightarrow \mathbb{R}^D, \forall r \in \Omega_R$, where $\Omega_R$ is the geometric rendezvous of the source and target.
- $\mathbf{M}$ is in general expensive to generate but cheap to apply
- For static $\mathbf{F}(s)$ and $\mathbf{G}(t)$, building $\mathbf{M}$ is a one-time, upfront cost

# The Rendezvous Algorithm

- Initially developed by the SIERRA team in the mid-2000's for parallel mesh-based data transfer [1]
- Creates a parallel topology map that can be used repeatedly for data transfer
- Map execution uses asynchronous strategy (posts and waits) with minimal messages
- Effectively $N * log(N)$ time complexity for parallel topology map generation
- Relies on the generation of a secondary decomposition of the source and target meshes with a geometric-based paritioning (RCB)

[1] S. Plimpton, B. Hendrickson, and J. Stewart, A parallel rendezvous algorithm for interpolation between multiple grids, Journal of Parallel and Distributed Computing, vol. 64, pp. 266276, 2004

# The Rendezvous Decomposition

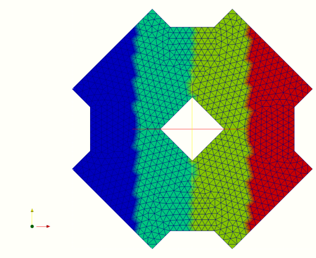


Figure: *Source mesh for 2D shared domain example.*

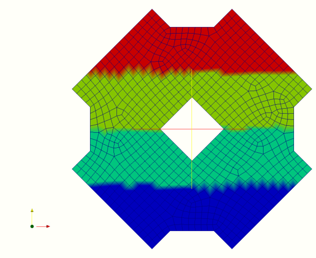


Figure: *Target mesh for 2D shared domain example.*

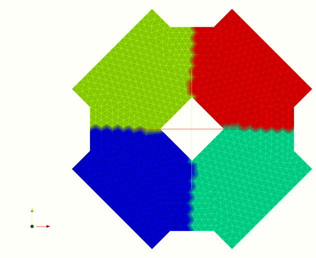


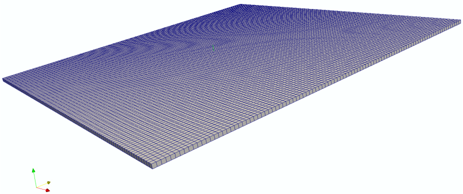
Figure: *Rendezvous decomposition for 2D shared domain example.*

# Searching the Rendezvous Decomposition

- Hierarchical parallel search tree
- Rendezvous decomposition provides parallel search
- kD-tree provides on-process proximity search
- Newton iterations provide final point location
- Results in reasonable scalability

# DTK Implementation Scaling Results

- Mesh-to-mesh transfer
- Worst case scenario study (all-to-all) with random points
- Qualitatively similar to the SIERRA results
- Largest test problems so far over 1.0E9 elements and 1.0E5 cores



# Strong Scaling

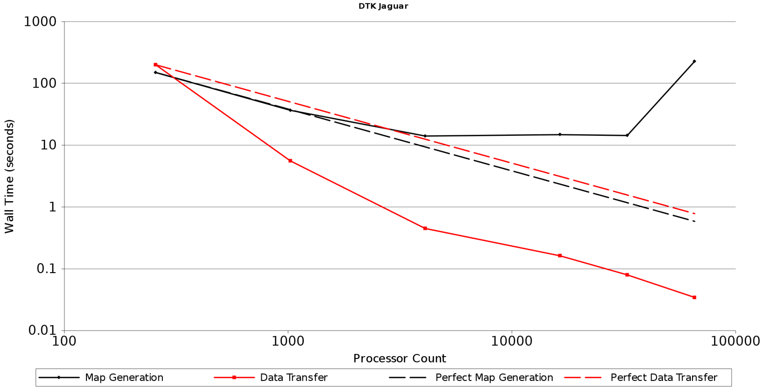



Figure: Strong scaling study results. The solid black curve reports the wall time to generate the mapping vs. number of processors while the solid red curve reports the wall time to transfer the data vs. number of processors. The dashed lines give perfect strong scaling the map generation (black) and the data transfer (red).

# Weak Scaling

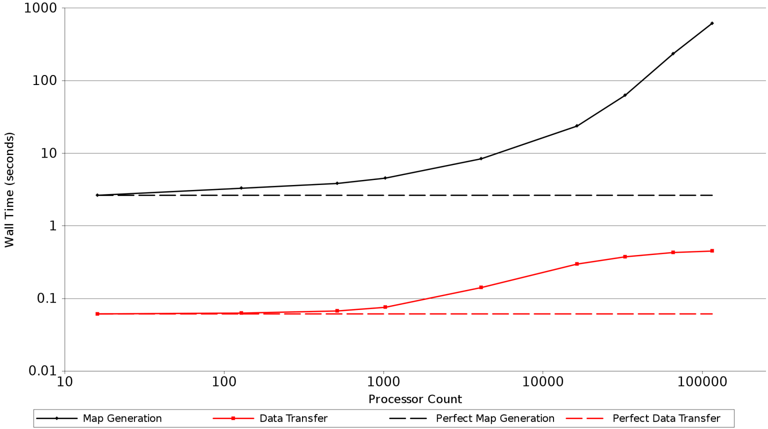



Figure: Weak scaling study results. The solid black curve reports the wall time to generate the mapping vs. number of processors while the solid red curve reports the wall time to transfer the data vs. number of processors. The dashed lines give perfect weak scaling the map generation (black) and the data transfer (red).

# Getting Data into DTK: Mesh

- Meshes are viewed as geometric structures
- A subset of total mesh information is needed:
    - Vertex coordinates
    - Element topology
    - Element connectivity
    - Connectivity permutation
- Parallel information is not required
- A communicator and global IDs are required

# Mesh Vertices and Elements

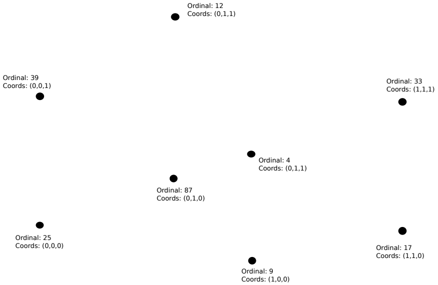



Figure: *Basic vertex description for a mesh.*

*Each vertex is required to have a unique global ID*

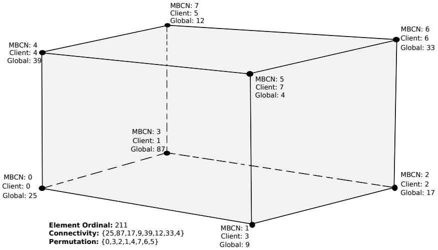



Figure: *Basic element description for a mesh.*

*Each element is required to have a unique global ID*

# Connectivity Permutation

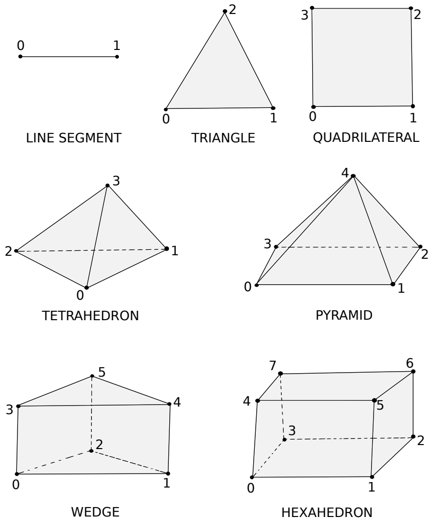



Figure: *Canonical vertex connectivity schemes for elements in DTK.*

- Allow for user specification of canonical ordering
- Permutation list specifies the variation in ordering for a specified topology
- Support for other mesh topologies currently not offered

# Meshes of Multiple Topologies

- All topologies in a mesh must be of the same dimension
- Each topology contained in a block
- Can have multiple blocks of the same topology

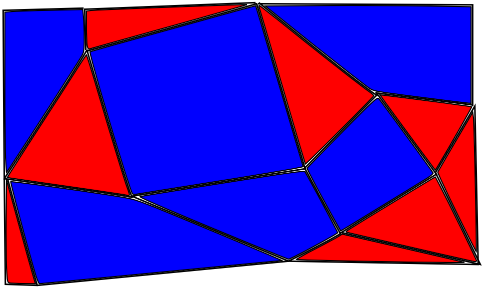


Figure: *Hybrid mesh example.*

*Quadrilaterals (blue) must be specified in a different mesh block than the triangles (red). Both blocks can contain the mutual mesh vertices that construct their elements.*

# Getting Data into DTK: Geometry

- DTK's perception of general geometric structures is primitive
- A more capable geometry engine required for advanced algorithms
- A subset of total geometry information is needed:
    - Centroid
    - Bounding box
    - Measure
    - Point inclusion
- Parallel information is not required
- A communicator and global IDs are required

# Getting Data into DTK: Field Evaluations

- Actual discretization of the field is not explicitly formulated
- Access to discretization of fields is generated through user code function evaluations at points in physical space:

$$\hat{f} \leftarrow \mathbf{F}(\hat{r}), \forall \hat{r} \in \Omega$$

- In the context of $\Omega$ discretized by a mesh, these evaluations can instead be written in terms of a single mesh element, $\omega \in \Omega$:

$$\hat{f} \leftarrow \mathbf{F}(\hat{r}), \forall \hat{r} \in \omega$$

# Getting Data into DTK: Field Evaluations

- What user code should expect:
  - A list of global element IDs that are on-process ($\omega$)
  - A corresponding list of point coordinates ($\hat{r}$)
- What user code should provide:
  - A list of the resulting function evaluations for each global element ID/point coordinate pair provided ($\hat{f}$)
- All data provided to user code will be local with respect to input data
- C++ inheritance (mix-in interface)

# Getting Data into DTK: Field Integrations

- Consider a measure-weighted integral:

$$f_\Omega = \frac{\int_\Omega \mathbf{F}(r)dr}{\int_\Omega dr}$$

- In the context of $\Omega$ discretized by a mesh, these evaluations can instead be written in terms of a single mesh element, $\omega \in \Omega$:

$$f_\omega = \int_\omega \mathbf{F}(r)dr$$

- The integral over $\Omega$ will be the measure-weighted summation of all element integrals:

$$f_\Omega = \frac{1}{m_\Omega} \sum_i f_{\omega_i},\ \forall \omega_i \in \Omega$$

- Element-wise spatial integrals generated through user code

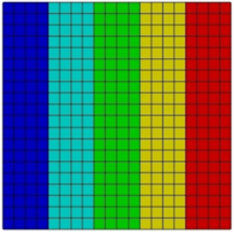
# Getting Data into DTK: Field Integrations

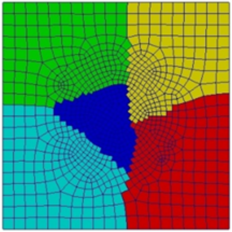
- What user code should expect:
  - A list of global element IDs that are on-process ($\omega$)
- What user code should provide:
  - A list of the resulting function integrations for each global element ID provided ($f_\omega$)
- All data provided to user code will be local with respect to input data
- C++ inheritance (mix-in interface)

# Getting Data into DTK: Target Data Space

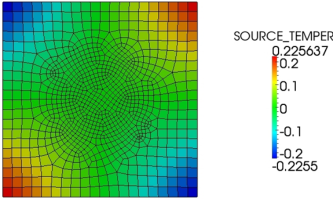
- What user code should provide:
  - Allocated memory block large enough to hold the resulting function evaluations or integrations for the specified local target objects
- What user code should expect:
  - No allocation of memory
  - A result of 0 if no evaluation or integration occurred for that object
- All data provided to user code will be local with respect to input data
- Implemented as array views (pointer and size)

# Standard Mesh-Based Rendezvous Map





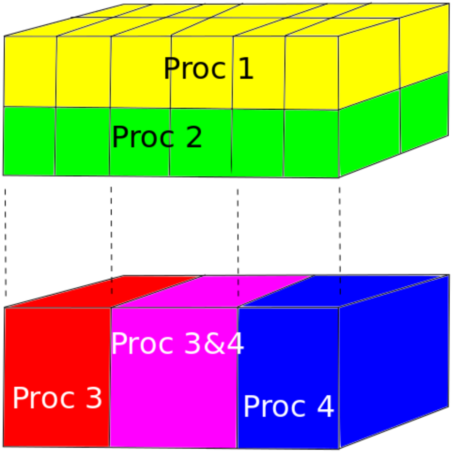
- Mesh-to-Mesh transfer [a]
- Used to move $\mathbf{F}(\hat{r})$ between meshes of arbitrary distribution
- Requires user code for evaluations in mesh elements



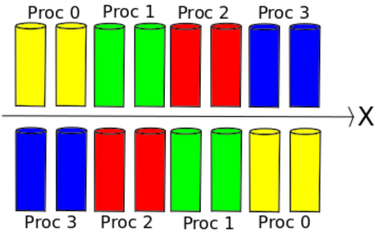

[a] Example provided by Roger Pawlowski

# Other Rendezvous-Based Maps: Integral Assembly

- Mesh-to-geometry transfer
- Used to assemble $f_\Omega$ with mesh and geometry of arbitrary distribution into measure-weighted integral
- The mesh is assumed conformal
- Requires user code for integrations in mesh elements
- See example/IntegralAssembly

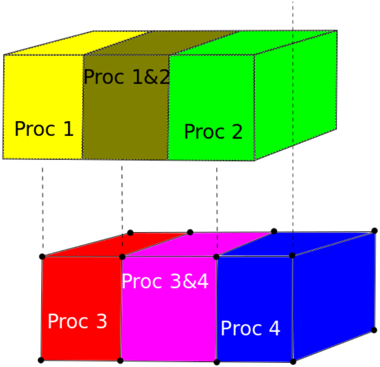


# Other Rendezvous-Based Maps: Geometry to Geometry




- Simple geometry-to-geometry transfer capability
- Geometries are assumed conformal
- Requires user code for evaluations in geometry
- See example/GeometryToGeometry

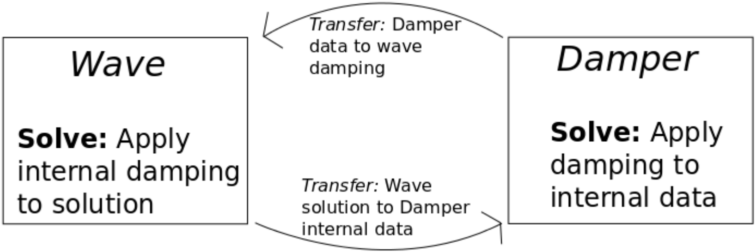
# Other Rendezvous-Based Maps: Geometry to Mesh




- Similar to mesh-based rendezvous
- Does not require a mesh, conceptual in this case
- Requires user code for evaluations in geometry
- See example/GeometryToMesh

# Super Simple Example

- examples/WaveDamper
- Using DTK in the context of Picard iteration
- Start with existing codes,
- Expose data to DTK



# Super Simple Example: Wave Code

- 1D code
- Initial conditions: $\mathbf{F}(x) = \cos(x)$

```
class Wave {
  private:
  Teuchos::RCP<const Teuchos::Comm<int> > comm;
  Teuchos::RCP<std::vector<double> > grid;
  Teuchos::RCP<std::vector<double> > data;
  Teuchos::RCP<std::vector<double> > damping;

  public:
  Wave( Teuchos::RCP<const Teuchos::Comm<int> > _comm,
  double x_min, double x_max, int num_x)
  {
    // Create the grid.
    // Set initial conditions.
  }

  void solve()
  { /* Apply the damping to the local data */ }
};
```

# Super Simple Example: Damper Code

- 1D code
- Initial conditions: none

```
class Damper
{
  private:
  Teuchos::RCP<const Teuchos::Comm<int> > comm;
  Teuchos::RCP<std::vector<double> > data;
  Teuchos::RCP<std::vector<double> > damping;
  Teuchos::RCP<std::vector<double> > grid;

  public:
  Damper( Teuchos::RCP<const Teuchos::Comm<int> > _comm,
  double x_min, double x_max, int num_x)
  { /* Create the grid. */ }

  void solve()
  { /* Apply damping to the local data. */ }
};
```