Design and Implementation of the Data Transfer Kit

Stuart R. Slattery
sslattery@wisc.edu

April 18, 2012

1 Introduction

For most multiphysics simulations, some form of data transfer between physics codes is required. In general, data transfer can be performed through an ad-hoc coupling of the codes with custom software designed for the specific simulation. However, it is useful to consider a component with an associated interface for data transfer operations. This common interface, to be fulfilled by each code that desires to participate in data transfer operations using this component, should reduce repeated effort and define generic and reusable software that can be applied to many forms of multiphysics simulation.

The data transfer package aims to define such an interface for coupling in addition to providing concrete implementations of algorithms and communication patterns for parallel data transfer using the information acquired through this interface. It should not be the aim of the data transfer package to eliminate other forms of multiphysics data transfer. Rather, it should attempt to define a useful interface that provides value by simplifying the data transfer process and reducing the barrier of entry for a physics code to participate in coupling.

This document discusses the design requirements of this package, named the Data Transfer Kit (DTK), the resulting interface specification, the implementation of a basic transfer algorithm using this interface, and instructions on building DTK.

2 Design Requirements for the Data Transfer Kit

The Data Transfer Kit should fulfill the following design requirements:

- Be able to correctly transfer data between multiple codes that have implemented the interface.
- Be independent of the underlying data structures of the physics codes using the interface.
- Be aware of the potential mesh entity structures that are associated with the fields being transferred (i.e. vertices, faces, elements, etc.).
- Be aware of the parallel communication objects that operate the codes being coupled.
- Be able to generate a parallel topology map using information made available through the interface.
- Be able to transfer multiple fields using the topology map and information made available through the interface.

3 Preliminary Design of the Data Transfer Kit

As part of the requirements listed in the previous section, it is necessary to define an interface for the Data Transfer Kit that will allow access to information held by each physics code being coupled in order to meet the coupling algorithm requirements in concrete data transfer implementations.

In its current form, the interface to the Data Transfer Kit exists as two abstract interfaces contained by the *DataSource* and the *DataTarget* classes. At least one of these interfaces should be fulfilled by each physics code that desires to participate in coupling under this framework by creating its own implementation of the *DataSource* interface for supplying data to the Data Transfer Kit and the *DataTarget* interface for receiving data from the Data Transfer Kit. By implementing these, a code will provide the necessary information to implement coupling algorithms.

The following sections provide the *DataSource* and *DataTarget* abstract interfaces. All methods defined by these interfaces are pure virtual and must therefore be defined by all implementations. Three template parameters are defined for the interfaces, the scalar data type to be transferred, the handle type associated with mesh entities (commonly int or long int), and the coordinate type for defining spatial locations (commonly double or float).

The Teuchos package of Trilinos is used for memory management, imposing the requirement on interface implementations to use these utilities. The underlying physics package need not implement these memory management tools, however, they must provide an interface implementation with these utilities. See [1] for more details on the Teuchos memory management scheme.

Communication is handled by the Teuchos abstract communication interface, allowing for various architectures to be supported. In the instance of MPI-based communication, any communication object implementing MPI primitives (e.g. MPI_SEND, MPI_RECEIVE, etc.) can be used.

As an initial attempt to create mesh aware data structures, a simple point object has been implemented as a container for coordinates and an associated identifier handle within the mesh subpackage. These interfaces also impose the requirement that this point data structure be used in the implemenations.

The following sections outline both interfaces and details on their requirements.

3.1 Data Source Interface

The *DataSource* interface consists of the following pure virtual methods to be implemented by the client application.

DataSource::get_source_comm() This method requires the client application to return a reference counting pointer (Teuchos::RCP objects) to a const Teuchos communicator object. For the case of an MPI implementation, this is not required to be MPI_COMM_WORLD.

DataSource::is_field_supported() This method checks whether or not a particular field, indicated by name with a string, is implemented for transfer by the client. Given this string, the client should return true if the field is supported, false if not.

DataSource::is_local_point() Given a point object, the client returns true if this point resides in the local domain and should therefore be mapped, and false if not. As this is the only method in the interface that supplies target points to the source application, it is recommended that if the point handles and/or coordinates are required by the client for data transfer, that they be cached during the implementation of this method.

DataSource::are_local_points() An addtional point query method, behaves identically to *DataSource::is_local_point()*, with the query operation performed for a series of points. A default implementation of this method is provided for those who only implement the *is_local_point* method.

DataSource::get_source_data() For a given field identified by a string, return a const persisting view (a Teuchos::ArrayRCP) of the data corresponding to the target points found in the local domain through the *get_points* method.

DataSource::get_global_data_data() For a given field identified by a string, return a single global data value.

The following listing provides the *DataSource* abstract interface.

```
template<class DataType, class HandleType, class CoordinateType, int DIM>
class DataSource : Teuchos::Describable
{
  typedef int                                         OrdinalType;
  typedef Point<DIM,HandleType,CoordinateType>        PointType;
  typedef Teuchos::Comm<OrdinalType>                  Communicator_t;
  typedef Teuchos::RCP<const Communicator_t>          RCP_Communicator;

  virtual RCP_Communicator get_source_comm() = 0;

  virtual bool is_field_supported( const std::string &field_name ) = 0;

  virtual bool is_local_point( const PointType &point ) = 0;

  virtual const Teuchos::ArrayRCP<bool>
  are_local_points( const Teuchos::ArrayRCP<PointType> points );

  virtual const Teuchos::ArrayRCP<DataType>
  get_source_data( const std::string &field_name ) = 0;
  virtual DataType
  get_global_source_data( const std::string &field_name ) = 0;
};
```

3.2 Data Target Interface

The *DataTarget* interface consists of the following pure virtual methods to be implemented by the client application.

DataTarget::get_target_comm() This method requires the client application to return a Teuchos::RCP to a const reference of its communicator object. For the case of an MPI implementation, this is not required to be MPI_COMM_WORLD.

DataTarget::is_field_supported() This method checks whether or not a particular field, indicated by name with a string, is implemented for transfer by the client. Given this string, the client should return true if the field is supported, false if not.

DataTarget::get_target_points() For a given field identified by a string, return a Teuchos::ArrayRCP (a persisting view) of all the points to which data should be mapped in the local target domain. All point handles are required to be globally unique.

DataTarget::get_target_data_space() For a given field identified by a string, return a non-const persisting view (Teuchos::ArrayRCP) of the data vector associated with the points provided by the *set_points* method. This view will be used to write data into the target.

DataTarget::set_global_target_data() For a given field identified by a string, a single global data value is provided.

The following listing provides the *DataTarget* abstract interface.

```
  template<class DataType, class HandleType, class CoordinateType, int DIM>
  class DataTarget : public Teuchos::Describable
  {
    typedef int                                         OrdinalType;
    typedef Point<DIM,HandleType,CoordinateType>        PointType;
    typedef Teuchos::Comm<OrdinalType>                  Communicator_t;
    typedef Teuchos::RCP<const Communicator_t>          RCP_Communicator;

    virtual RCP_Communicator get_target_comm() = 0;

    virtual bool is_field_supported( const std::string &field_name ) = 0;

    virtual const Teuchos::ArrayRCP<PointType>
    get_target_points( const std::string &field_name ) = 0;

    virtual Teuchos::ArrayRCP<DataType>
    get_target_data_space( const std::string &field_name ) = 0;

    virtual void set_global_target_data( const std::string &field_name,
    const DataType &data ) = 0;
  };
```

3.3 Copy Operator

The relationship between a particular *DataSource* and *DataTarget* implementation is contained by the *CopyOperator* object. Both mapping and copy operations between the *DataSource* and *DataTarget* are handled by the *CopyOperator*. The following methods are available for the *CopyOperator* object.

CopyOperator::CopyOperator() The constructor for this object requires the communicator on which to operate. At a minimum, this should correspond to the union of the target and source communicators. Again, for MPI implementations this is not limited to MPI_COMM_WORLD. Rather, it is only required that this communicator span the process ID space that encompasses both communicators provided by the source and target interfaces. In addition, a string defining the name of the field, reference counting pointers to the two interface implementations, and an optional boolean indicating whether or not this is a scalar field. The default value of this boolean is false, indicating that the field is distributed.

CopyOperator::create_copy_mapping() Generate the parallel mapping between the source and target applications for the copy operation. This method has no arguments but is required to be called before copy operations can be completed.

CopyOperator::copy() Copy the data from the source application to the target application. This method has no arguments.

The following listing gives an example at the driver level of how the *CopyOperator* object is used to transfer data between the two interfaces. In this example, the scalar data type to be transferred is double, handle type is int, and coordinate type is float.

```
Teuchos::RCP<DataSource<double,int,float> > source =
Teuchos::rcp( new DataSource_Implementation<double,int,float>() );

Teuchos::RCP<DataTarget<double,int,float> > target =
Teuchos::rcp( new DataTarget_Implementation<double,int,float>() );

CopyOperator<double,int,float> copy_op( MPI_COMM_WORLD,
                                        "SOURCE_DISTRIBUTED_FIELD",
                                        "TARGET_DISTRIBUTED_FIELD",
                                        source,
                                        target );

copy_op.create_copy_mapping();
copy_op.copy();
```

4 Consistent Finite Element Interpolation

The initial implementation of the Data Transfer Kit is focused on performing consistent finite element interpolation of the following form as outlined in [2] [3]. We define ϕ_i to be the shape functions discretizing the source function f and ψ_i to be the shape functions discretizing the target function g. Given a set of target points, t, associated with g and its mesh, and a set of source points s associated with f and its mesh, we evaluate f such that $g(t) = \sum_i \phi_i(t) f(s)$. Here, we evaluate all components of the source basis ϕ at the target point and apply to it the source function value at each component location (typically the nodes of the mesh cells). It should be noted however, that this scheme is not strictly conservative as outlined in Appendix C of [3]. We therefore provide the set of *global_data* methods as a means to transfer global scalar data which can then be applied in a rebalance to enforce global conservation of quantity (but not local conservation).

From a design standpoint, the interface methods reflect the requirements to perform this computation. The method *get_target_points* populates t while *is_local_point* method determines the cell location (both geometric and parallel) of each component in t in order to perform the source basis function evaluation $\phi_i(t)$ in the summation. The *get_source_data* method provides $\sum_i \phi_i(t) f(s)$ for each target point in t, while *get_target_data_space* provides the space $g(t)$ in which to apply the function evaluation data for each target point.

4.1 Mapping Algorithm Implementation

The following algorithm is used to generate a point based mapping between between the source and target applications.

- Get the target points from the target application to populate the target point set.

- Extract the globally unique handles from the target points.
- Build a distributed map of the target data vector using the handles as indices.
- Do a global reduction across all nodes to compute the maximum number of local target points.
- On each target node, build a buffer of points to send to the source nodes. This buffer will be of the length computed by the global reduction and padded with null points.
- For (all target nodes):
  - Broadcast the buffer of target points for the current target node to all other nodes.
  - For (all points in the broadcasted buffer):
    - Query the source application domain with the point.
    - If (the point is found in the local source application domain):
      - Add the point handle to a distributed list.

      EndIf

    EndFor

  EndFor
- Build a distributed map of the source data vector using the distributed list of handles found in the source domain as indices.
- Setup a communication plan to transfer data between the source map distribution and the target map distribution.

Per the requirement of the target application to provide globally unique handles for all of its points, the end result of this mapping is a distributed object with knowledge of how the target data is distributed via the handle indices, and how the source data is distributed (most likely differently) using those same handle indices. Once these two distributions are known, the data transfer operation is simply a copy of data from the source to the target with each piece of source data sent to the target destination containing its handle index determined by the communication plan.

5 Data Transfer Kit Error Handling Policy

The Data Transfer Kit will rely on C++ exceptions as the primary means of propagating error information to the user. Per the discussion in [4], all runtime exceptions thrown by DTK will inherit from std::runtime_error, allowing users to isolate errors of this type with try/catch blocks. Internally, these runtime errors are divided into three categories; precondition exceptions that arise from a function's inability to meet preconditions, postcondition exceptions resulting from a function's failure to achieve a postcondition, and invariant exceptions resulting from a function's inability to establish or maintain an invariant. Throughout DTK, a user should expect these exceptions to be used to test these three fundamental conditions and for runtime exceptions to be thrown if they are violated. In addition, when an exception is thrown, a user should expect DTK to provide a message detailing why the exception was thrown.

For developers implementing the data transfer interfaces defined within DTK, it is expected that these exceptions are used to verify the three runtime conditions within their implementation. Doing so will then present a consistent exception scheme at the boundaries of DTK for users implementing the DTK API that are neither DTK developers or DTK interface developers. In addition, throwing these exceptions in the interface implementations will facilitate error propagation from the interface implementations to the user through the various API calls.

Internally, DTK relies on the Trilinos packages Teuchos and Tpetra, both of which throw C++ exceptions through their own schemes. Calls to these libraries within DTK will not be isolated with try/catch blocks. Instead, their exceptions will be propagated through the API boundary.

6 Obtaining the Data Transfer Kit

The Data Transfer Kit is version controlled with a git software repository residing on the casl-dev server. Changes to this repository are communicated to a CASL hosted and moderated mailing list. Access requests to this list can be made at *http://casl-dev.ornl.gov/mailman/listinfo/coupler-infrastructure*. In addition, this repository includes the Data Transfer Kit into the VERA continuous integration and nightly build process. Per the discussion in the TriBITS lifecycle model [5], the Data Transfer Kit is designated as secondary stable (SS).

To begin, the VERA repository is required as the base infrastructure for building and executing the Data Transfer Kit as a VERA component. Due to its direct dependencies on the Trilinos packages Teuchos and Tpetra, the second step in obtaining the Data Transfer Kit is to obtain Trilinos. Using the fissile four development machines and the associated development environment, the following steps are required to build VERA with the Data Transfer Kit. For purposes of testing this documentation, the machine u233 was used.

git clone /casl-dev/git-root/VERA VERA Clone a copy of the VERA git repository.

cd VERA Move into the VERA directory.

git clone /casl-dev/git-root/Trilinos Trilinos Clone a copy of the Trilinos git repository into the VERA home directory.

git clone /casl-dev/git-root/DataTransferKit DataTransferKit Clone a copy of the DataTransferKit repository into the VERA home directory.

7 Building the Data Transfer Kit

Once the Trilinos and DataTransferKit repositories have been cloned, both are ready to be configured with VERA. The Data Transfer Kit uses the Tribits build system based on cmake and configures as a VERA TPL.

7.1 Configure

Consider an out-of-source build in a */vera_build* directory. The following shell script, also found in the location */DataTransferKit/doc/build_notes/sample_cmake_configure.sh*, configures Trilinos and the DataTransferKit for a parallel debug build using the default MPI implementation in the fissile four development environment. To date, the Data Transfer Kit has been tested with both OpenMPI and MPICH2 implementations of the MPI specification on shared and distributed memory hardware.

```
#!/bin/bash

cmake \
-D CMAKE_INSTALL_PREFIX:PATH=/vera_build              \
-D CMAKE_BUILD_TYPE:STRING=DEBUG                      \
-D CMAKE_VERBOSE_MAKEFILE:BOOL=ON                     \
-D TPL_ENABLE_MPI:BOOL=ON                             \
-D VERA_EXTRA_REPOSITORIES="Trilinos;DataTransferKit" \
-D Teuchos_ENABLE_EXTENDED:BOOL=ON                    \
-D VERA_ENABLE_DataTransferKit:BOOL=ON                \
-D DataTransferKit_ENABLE_TESTS:BOOL=ON               \
-D DataTransferKit_ENABLE_EXAMPLES:BOOL=ON            \
/VERA
```

Aside from a standard VERA configuration with Trilinos, we designate that the DataTransferKit is an extra VERA repository, the Teuchos package in Trilinos should be extended to enable the communication utilities, and the Data Transfer Kit should be enabled by VERA with tests and examples on.

7.2 Build

After configuration, *make* will build the library while *make install* will install it in the specified prefix location.

7.3 Test

Unit tests in both serial and parallel have been implemented using the Teuchos unit testing harness and are incorporated into the Tribits build. Using the configuration above, after building the tests can be invoked with ctest. If successful, you should see an output like that presented below. Note below that the WaveDamper example to be later discussed was executed as well.

```
stuart@beaker:~/software/builds/VERA$ ctest
Test project /home/stuart/software/builds/VERA
    Start 1: DataTransferKitCore_CommIndexer_test_MPI_4
1/7 Test #1: DataTransferKitCore_CommIndexer_test_MPI_4 .........   Passed    1.72 sec
    Start 2: DataTransferKitCore_Point_test_MPI_4
2/7 Test #2: DataTransferKitCore_Point_test_MPI_4 ...............   Passed    1.13 sec
    Start 3: DataTransferKitCore_Interfaces_test_MPI_4
3/7 Test #3: DataTransferKitCore_Interfaces_test_MPI_4 ..........   Passed    1.12 sec
    Start 4: DataTransferKitCore_CopyOperator_test_MPI_4
4/7 Test #4: DataTransferKitCore_CopyOperator_test_MPI_4 ........   Passed    1.28 sec
    Start 5: DataTransferKitCore_Advanced_Transfer_test_MPI_4
5/7 Test #5: DataTransferKitCore_Advanced_Transfer_test_MPI_4 ...   Passed    1.16 sec
    Start 6: DataTransferKitCore_Zero_Point_Proc_test_MPI_4
6/7 Test #6: DataTransferKitCore_Zero_Point_Proc_test_MPI_4 .....   Passed    1.15 sec
    Start 7: DataTransferKitCore_WaveDamperExample_MPI_4
7/7 Test #7: DataTransferKitCore_WaveDamperExample_MPI_4 ........   Passed    1.23 sec

100% tests passed, 0 tests failed out of 7

Label Time Summary:
DataTransferKit    =   8.79 sec

Total Test time (real) =  11.87 sec
```

8 Using the Data Transfer Kit

To demonstrate using the Data Transfer Kit in a multiphysics simulation, a simple example was constructed to show both implementations of the *DataSource* and *DataTarget* interfaces specified above and a multiphysics driver utilizing the Data Transfer Kit to perform data transfer in an iterative manner.

8.1 WaveDamper Example

Provided as an example in the DataTransferKit repository and enabled in the above configuration, the WaveDamper example exists in the directory */DataTransferKit/example/WaveDamper*. Two simple codes are coupled in the simulation; a code called Wave that computes a one dimensional cosine shape over the parallel domain and another code called Damper that computes the amount by which the amplitude of an incoming function should be decreased. They are coupled through a Picard iteration in the following fashion until the amplitude of the wave is dampened below a specified tolerance.

```
Wave computes its initial conditions.

while the amplitude of the wave computed by Wave is > tolerance:

    Wave transfers its solution to Damper.
```

```
Damper computes the amount by which the amplitude of the Wave
solution should be decreased.

Damper transfers the computed dampening back to Wave.

Wave applies the dampening to compute a new solution.
```

Here, the convergence tolerance on the wave amplitude is specified as 1.0E-6 in the WaveDamper implementation.

8.1.1 Wave Implementation

The Wave code was implemented as shown in the following listings. Note the methods included to provide access to various members relating to data and the grid that have been added. These are similar to the types of methods that will be required during the refactoring process necessary for using the Data Transfer Kit. It should also be noted that the Data Transfer Kit does not place any requirements on solve methods. For the Wave code solve, note that we are additionally returning the L2 norm of the vector we are operating on. In the iteration sequence presented in the driver code below, the magnitude of the L2 norm will be used as a stopping criteria.

```
#include <vector>

#include "Teuchos_RCP.hpp"
#include "Teuchos_Comm.hpp"

class Wave
{
  private:

  Teuchos::RCP<const Teuchos::Comm<int> > comm;
  std::vector<double> grid;
  std::vector<double> f;
  std::vector<double> damping;

  public:

  Wave(Teuchos::RCP<const Teuchos::Comm<int> > _comm,
  double x_min,
  double x_max,
  int num_x);

  ~Wave();

  // Get the communicator.
  Teuchos::RCP<const Teuchos::Comm<int> > get_comm()
  {
    return comm;
  }

  // Get a const reference to the local grid.
  const std::vector<double>& get_grid()
  {
    return grid;
  }

  // Get a reference to the local data.
  std::vector<double>& get_f()
  {
    return f;
  }

  // Apply the damping to the local data structures from an external
  // source.
  std::vector<double>& set_damping()
  {
    return damping;
  }

  // Solve the local problem and return the l2 norm of the local residual.
  double solve();
};
```

```
#include "Wave.hpp"

#include <algorithm>
#include <cmath>

Wave::Wave(Teuchos::RCP<const Teuchos::Comm<int> > _comm,
double x_min,
double x_max,
int num_x)
: comm(_comm)
{
  // Create the grid.
  grid.resize(num_x);
  double x_size = (x_max - x_min) / (num_x);

  std::vector<double>::iterator grid_iterator;
  int i = 0;

  for (grid_iterator = grid.begin();
  grid_iterator != grid.end();
  ++grid_iterator, ++i)
  {
    *grid_iterator = i*x_size + x_min;
  }

  // Set initial conditions.
  damping.resize(num_x);
  std::fill(damping.begin(), damping.end(), 0.0);
  f.resize(num_x);
  std::vector<double>::iterator f_iterator;
  for (f_iterator = f.begin(), grid_iterator = grid.begin();
  f_iterator != f.end();
  ++f_iterator, ++grid_iterator)
  {
    *f_iterator = cos( *grid_iterator );
  }
}

Wave::~Wave()
{ /* ... */ }

double Wave::solve()
{
  // Apply the dampened component.
  double l2_norm_residual = 0.0;
  double f_old = 0.0;
  std::vector<double>::iterator f_iterator;
  std::vector<double>::const_iterator damping_iterator;
  for (f_iterator = f.begin(), damping_iterator = damping.begin();
  f_iterator != f.end();
  ++f_iterator, ++damping_iterator)
  {
    f_old = *f_iterator;
    *f_iterator -= *damping_iterator;
    l2_norm_residual += (*f_iterator - f_old)*(*f_iterator - f_old);
  }

  // Return the l2 norm of the local residual.
  return pow(l2_norm_residual, 0.5);
}
```

8.1.2 Damper Implementation

The following listings give the implementation of the Damper code. Again, note the methods used to provide access to the various data structures needed by the Data Transfer Kit. These methods will be used in the interface implementations given by the following sections.

```
#include <vector>

#include "Teuchos_RCP.hpp"
#include "Teuchos_Comm.hpp"
```

```cpp
class Damper
{
  private:

  Teuchos::RCP<const Teuchos::Comm<int> > comm;
  std::vector<double> wave_data;
  std::vector<double> damping;
  std::vector<double> grid;

  public:

  Damper(Teuchos::RCP<const Teuchos::Comm<int> > _comm,
  double x_min,
  double x_max,
  int num_x);

  ~Damper();

  // Get the communicator.
  Teuchos::RCP<const Teuchos::Comm<int> > get_comm()
  {
    return comm;
  }

  // Get a reference to the local damping data.
  std::vector<double>& get_damping()
  {
    return damping;
  }

  // Get a reference to the local grid.
  std::vector<double>& get_grid()
  {
    return grid;
  }

  // Get the wave data to apply damping to from an external source.
  std::vector<double>& set_wave_data()
  {
    return wave_data;
  }

  // Apply damping to the local problem.
  void solve();
};
```

```cpp
#include "Damper.hpp"

Damper::Damper(Teuchos::RCP<const Teuchos::Comm<int> > _comm,
double x_min,
double x_max,
int num_x)
: comm(_comm)
{
  // Create the grid.
  grid.resize(num_x);
  double x_size = (x_max - x_min) / (num_x);

  std::vector<double>::iterator grid_iterator;
  int i = 0;

  for (grid_iterator = grid.begin();
  grid_iterator != grid.end();
  ++grid_iterator, ++i)
  {
    *grid_iterator = i*x_size + x_min;
  }

  // Set initial conditions.
  damping.resize(num_x);
  wave_data.resize(num_x);
}

Damper::~Damper()
{ /* ... */ }

// Apply damping to the local problem.
```

```
void Damper::solve()
{
  std::vector<double>::iterator damping_iterator;
  std::vector<double>::const_iterator wave_data_iterator;
  for (damping_iterator = damping.begin(),
  wave_data_iterator = wave_data.begin();
  damping_iterator != damping.end();
  ++damping_iterator, ++wave_data_iterator)
  {
    *damping_iterator = *wave_data_iterator / 2;
  }
}
```

8.1.3 Wave Interface Implementations

As given above by the algorithm used to couple Wave to Damper, two way transfer is required and therefore both codes must implement the *DataSource* and *DataTarget* in order for this scheme to be possible. The following listings in this section give the interface implementations for the Wave code. For both interfaces, the scalar transfer methods were implemented to perform no operations as this example does not make use of them. Both implementations, though templated to inherit from the pure virtual interface specifications, do not use the template parameters. Rather, they explicitly specify in the typdef block elements of type double to be transferred, int to be used for point handles, and doubles to be used for coordinates.

To begin, the *DataSource* implementation listing for the Wave code has several elements that should be addressed. First, the constructor in this implementation, and that of the *DataTarget* implementation, takes a pointer to the wave object that it will operate on. For the *get_comm()* methods, a simple redirection is used to grab the communicator from the local Wave object.

For all transfers under this interface, the name of the field being transferred is used to identify the appropriate operations to perform in the interface. The *Wave_Data_Source* implementation will serve as the source for the WAVE_FIELD and therefore the *is_field_supported()* method designates this. In addition, all other interface methods operating on a specific field will check for this name before doing operations.

The *is_local_point()* method checks the incoming point against the local grid. If it is found to be a local grid point, true is returned. Finally, the *get_source_data()* method returns a view of the Wave code solution, provided by a refactor method *get_f()*.

```
namespace DataTransferKit {

// DataSource interface implementation for the Wave code.
template<class DataType, class HandleType, class CoordinateType, int DIM>
class Wave_DataSource
    : public DataSource<DataType,HandleType,CoordinateType,DIM>
{
  public:

    typedef int                                            OrdinalType;
    typedef Point<1,HandleType,CoordinateType>               PointType;
    typedef Teuchos::Comm<OrdinalType>                     Communicator_t;
    typedef Teuchos::RCP<const Communicator_t>             RCP_Communicator;
    typedef Teuchos::RCP<Wave>                             RCP_Wave;

  private:

    // Wave object to operate on.
    RCP_Wave wave;

  public:

    Wave_DataSource(RCP_Wave _wave)
    : wave(_wave)
    { /* ... */ }

    ~Wave_DataSource()
    { /* ... */ }

    RCP_Communicator get_source_comm()
    {
    return wave->get_comm();
    }
    bool is_field_supported(const std::string &field_name)
    {
    bool return_val = false;

    if (field_name == "WAVE_SOURCE_FIELD")
    {
        return_val = true;
    }

    return return_val;
    }

    bool is_local_point(const PointType &test_point)
    {
    bool return_val = false;

    if ( std::find(wave->get_grid().begin(),
                   wave->get_grid().end(),
                   test_point.getCoords()[0] )
         != wave->get_grid().end() )
    {
        return_val = true;
    }

    return return_val;
    }

    const Teuchos::ArrayView<DataType>
    get_source_data(const std::string &field_name)
    {
    Teuchos::ArrayView<DataType> return_view;

    if ( field_name == "WAVE_SOURCE_FIELD" )
    {
        return_view = Teuchos::ArrayView<DataType>( wave->get_f() );
    }

    return return_view;
    }

    DataType get_global_source_data(const std::string &field_name)
    {
    DataType return_val = 0.0;

    return return_val;
    }
};
} // end namespace DataTransferKit
```

The *DataTarget* interface implementation for the Wave code follows the same principles as the *DataSource* implementation given by the previous listing. In this case, the Wave code is acting as a target for the DAMPER_FIELD and therfore all methods operating by field name check for this field. In addition, the *get_target_points* method recasts the one dimensional grid used by the Wave code as a vector of three dimensional point objects, storing them as member data. Again, the refactor methods are used here to provide views into the private data structures of Wave.

```
namespace DataTransferKit {

// DataTarget interface implementation for the Wave code.
template<class DataType, class HandleType, class CoordinateType, int DIM>
class Wave_DataTarget
    : public DataTarget<DataType,HandleType,CoordinateType,DIM>
{
  public:

    typedef int                                         OrdinalType;
    typedef Point<1,HandleType,CoordinateType>          PointType;
    typedef Teuchos::Comm<OrdinalType>                  Communicator_t;
    typedef Teuchos::RCP<const Communicator_t>          RCP_Communicator;
    typedef Teuchos::RCP<Wave>                          RCP_Wave;

  private:

    RCP_Wave wave;
    std::vector<PointType> local_points;

  public:

    Wave_DataTarget(RCP_Wave _wave)
    : wave(_wave)
    { /* ... */ }

    ~Wave_DataTarget()
    { /* ... */ }

    RCP_Communicator get_target_comm()
    {
    return wave->get_comm();
    }

    bool is_field_supported(const std::string &field_name)
    {
    bool return_val = false;

    if (field_name == "DAMPER_TARGET_FIELD")
    {
        return_val = true;
    }

    return return_val;
    }

    const Teuchos::ArrayView<PointType>
    get_target_points(const std::string &field_name)
    {
    Teuchos::ArrayView<PointType> return_view;

    if ( field_name == "DAMPER_TARGET_FIELD" )
    {
        local_points.clear();
        Teuchos::ArrayView<const DataType> local_grid( wave->get_grid() );
        typename Teuchos::ArrayView<DataType>::const_iterator grid_it;
        int n = 0;
        int global_handle;
        for (grid_it = local_grid.begin();
         grid_it != local_grid.end();
         ++grid_it, ++n)
        {
        global_handle = wave->get_comm()->getRank() *
                local_grid.size() + n;
        local_points.push_back(
            point( global_handle, *grid_it) );
        }
        return_view = Teuchos::ArrayView<PointType>(local_points);
    }

    return return_view;
    }

    Teuchos::ArrayView<DataType>
    get_target_data_space(const std::string &field_name)
    {
    Teuchos::ArrayView<DataType> return_view;

    if ( field_name == "DAMPER_TARGET_FIELD" )
    {
        return_view = Teuchos::ArrayView<DataType>( wave->set_damping() );
    }

    return return_view;
    }

    void set_global_target_data(const std::string &field_name,
                const DataType &data)
    { /* ... */ }
};
} // end namespace DataTransferKit
```

8.1.4 Damper Interface Implementations

This section gives the two interface implementations for the Damper code. The same principles used in the implementations of the Wave interfaces also apply here. Again, the *Damper_Data_Source* implementation will only provide data on points that exist on its grid. If they do not, no data will be applied. As the

Damper is the source for the DAMPER_FIELD and a target for the WAVE_FIELD, these string values are used accordingly within the implementations.

```
namespace DataTransferKit {

// DataSource interface implementation for the Damper code.
template<class DataType, class HandleType, class CoordinateType, int DIM>
class Damper_DataSource
    : public DataSource<DataType,HandleType,CoordinateType,DIM>
{
  public:

    typedef int                                         OrdinalType;
    typedef Point<DIM,HandleType,CoordinateType>        PointType;
    typedef Teuchos::Comm<OrdinalType>                  Communicator_t;
    typedef Teuchos::RCP<const Communicator_t>          RCP_Communicator;
    typedef Teuchos::RCP<Damper>                        RCP_Damper;

  private:

    // Damper object to operate on.
    RCP_Damper damper;

  public:

    Damper_DataSource(RCP_Damper _damper)
    : damper(_damper)
    { /* ... */ }

    ~Damper_DataSource()
    { /* ... */ }

    RCP_Communicator get_source_comm()
    {
    return damper->get_comm();
    }

    bool is_field_supported(const std::string &field_name)
    {
    bool return_val = false;

    if (field_name == "DAMPER_SOURCE_FIELD")
    {
        return_val = true;
    }

    return return_val;
    }

    bool is_local_point(const PointType &test_point)
    {
    bool return_val = false;

    if ( std::find(damper->get_grid().begin(),
               damper->get_grid().end(),
               test_point.getCoords()[0] )
         != damper->get_grid().end() )
    {
        return_val = true;
    }

    return return_val;
    }

    const Teuchos::ArrayView<DataType>
    get_source_data(const std::string &field_name)
    {
    Teuchos::ArrayView<DataType> return_view;

    if ( field_name == "DAMPER_SOURCE_FIELD" )
    {
        return_view = Teuchos::ArrayView<DataType>( damper->get_damping() );
    }

    return return_view;
    }

    DataType get_global_source_data(const std::string &field_name)
    {
```

```
    DataType return_val = 0.0;

    return return_val;
    }
};
} // end namespace DataTransferKit
```

As in the *Wave_Data_Target* implementation, the Damper implementation utilizes the *Point* container to provide the expected representation of its grid to the interfaces. Note that the point container, and thus the interfaces, is templated on spatial dimension.

```
namespace DataTransferKit {

// DataTarget interface implementation for the Damper code.
template<class DataType, class HandleType, class CoordinateType, int DIM>
class Damper_DataTarget
    : public DataTarget<DataType, HandleType, CoordinateType,DIM>
{
  public:

    typedef int                                       OrdinalType;
    typedef Point<1,HandleType,CoordinateType>        PointType;
    typedef Teuchos::Comm<OrdinalType>                Communicator_t;
    typedef Teuchos::RCP<const Communicator_t>        RCP_Communicator;
    typedef Teuchos::RCP<Damper>                      RCP_Damper;

  private:

    RCP_Damper damper;
    std::vector<PointType> local_points;

  public:

    Damper_DataTarget(RCP_Damper _damper)
    : damper(_damper)
    { /* ... */ }

    ~Damper_DataTarget()
    { /* ... */ }

    RCP_Communicator get_target_comm()
    {
    return damper->get_comm();
    }

    bool is_field_supported(const std::string &field_name)
    {
    bool return_val = false;

    if (field_name == "WAVE_TARGET_FIELD")
    {
        return_val = true;
    }

    return return_val;
    }

    const Teuchos::ArrayView<PointType>
    get_target_points(const std::string &field_name)
    {
    Teuchos::ArrayView<PointType> return_view;

    if ( field_name == "WAVE_TARGET_FIELD" )
    {
        local_points.clear();
        Teuchos::ArrayView<const DataType> local_grid( damper->get_grid() );
        typename Teuchos::ArrayView<DataType >::const_iterator grid_it;
        int n = 0;
        int global_handle;
        for (grid_it = local_grid.begin();
         grid_it != local_grid.end();
         ++grid_it, ++n)
        {
        global_handle = damper->get_comm()->getRank() *
                local_grid.size() + n;
        local_points.push_back(
            point( global_handle, *grid_it) );
        }
        return_view = Teuchos::ArrayView<PointType>(local_points);
    }

    return return_view;
    }

    Teuchos::ArrayView<DataType>
    get_target_data_space(const std::string &field_name)
    {
    Teuchos::ArrayView<DataType> return_view;

    if ( field_name == "WAVE_TARGET_FIELD" )
    {
        return_view = Teuchos::ArrayView<DataType>( damper->set_wave_data() );
    }

    return return_view;
    }

    void set_global_target_data(const std::string &field_name,
                                const DataType &data)
    { /* ... */ }
};
} // end namespace DataTransferKit
```

8.1.5 Coupled Driver

Once the interfaces have been implemented for both codes, the infrastructure is in place to do a simple coupled problem. The following listing gives the multiphysics driver used in the WaveDamper problem.

After parallel initialization, a domain from 0.0 to 5.0 is distributed in parallel across each process for each code. The bounds of the computed local domain are applied in the constructor for each code. Following this, the WAVE_FIELD is setup for transfer. An instance of the *DataSource* interface is created from the *Wave_Data_Source* implementation and *DataTarget* interface is created from the *Damper_Data_Target* implementation. A *DataField* is then constructed to transfer the WAVE_FIELD from the Wave code to the Damper code. The same process is repeated for the DAMPER_FIELD.

Once the various Data Transfer Kit components have been initialized, an iteration loop is setup to run until either the L2 norm of the wave amplitude computed by the Wave code is less than 1.0E-6 or a specified maximum number of iterations has been reached, each time executing the algorithm specified by the picard iteration pseudocode above.

```
#include <iostream>

#include "Wave.hpp"
#include "Wave_Source.hpp"
#include "Wave_Target.hpp"
#include "Damper.hpp"
#include "Damper_Source.hpp"
#include "Damper_Target.hpp"

#include <DataTransferKit_DataSource.hpp>
#include <DataTransferKit_DataTarget.hpp>
#include <DataTransferKit_CopyOperator.hpp>

#include "Teuchos_RCP.hpp"
#include "Teuchos_CommHelpers.hpp"
#include "Teuchos_DefaultComm.hpp"
#include "Teuchos_GlobalMPISession.hpp"

//---------------------------------------------------------------------------//
// Main function driver for the coupled Wave/Damper problem.
int main(int argc, char* argv[])
{
    // Setup communication.
    Teuchos::GlobalMPISession mpiSession(&argc,&argv);
    Teuchos::RCP<const Teuchos::Comm<int> > comm =
    Teuchos::DefaultComm<int>::getComm();

    // Set up the parallel domain.
    double global_min = 0.0;
    double global_max = 5.0;
    int myRank = comm->getRank();
    int mySize = comm->getSize();
    double local_size = (global_max - global_min) / mySize;
    double myMin = myRank*local_size + global_min;
    double myMax = (myRank+1)*local_size + global_min;

    // Setup a Wave.
    Teuchos::RCP<Wave> wave =
    Teuchos::rcp( new Wave(comm, myMin, myMax, 10) );

    // Setup a Damper.
    Teuchos::RCP<Damper> damper =
    Teuchos::rcp( new Damper(comm, myMin, myMax, 10) );

    // Setup a Wave Data Source for the wave field.
    Teuchos::RCP<DataTransferKit::DataSource<double,int,double,1> >
    wave_source = Teuchos::rcp(
        new DataTransferKit::Wave_DataSource<double,int,double,1>(wave) );

    // Setup a Damper Data Target for the wave field.
    Teuchos::RCP<DataTransferKit::DataTarget<double,int,double,1> >
    damper_target = Teuchos::rcp(
        new DataTransferKit::Damper_DataTarget<double,int,double,1>(damper) );

    // Setup a copy operator for the wave field.
    DataTransferKit::CopyOperator<double,int,double,1>
    wave_field_op( comm,
                   "WAVE_SOURCE_FIELD",
                   "WAVE_TARGET_FIELD",
                   wave_source,
                   damper_target );

    // Setup a Damper Data Source for the damper field.
    Teuchos::RCP<DataTransferKit::DataSource<double,int,double,1> >
    damper_source = Teuchos::rcp(
        new DataTransferKit::Damper_DataSource<double,int,double,1>(damper) );

    // Setup a Wave Data Target for the damper field.
    Teuchos::RCP<DataTransferKit::DataTarget<double, int, double,1> >
    wave_target = Teuchos::rcp(
        new DataTransferKit::Wave_DataTarget<double,int,double,1>(wave) );

    // Setup a copy operator for the damper field.
    DataTransferKit::CopyOperator<double,int,double,1>
    damper_field_op( comm,
                 "DAMPER_SOURCE_FIELD",
                 "DAMPER_TARGET_FIELD",
                 damper_source,
                 wave_target );

    // Iterate between the damper and wave until convergence.
    double local_norm = 0.0;
    double global_norm = 1.0;
    int num_iter = 0;
    int max_iter = 100;

    // Create the mapping for the wave field.
    wave_field_op.create_copy_mapping();

    // Create the mapping for the damper field.
    damper_field_op.create_copy_mapping();

    while( global_norm > 1.0e-6 && num_iter < max_iter )
    {
    // Transfer the wave field.
    wave_field_op.copy();

    // Damper solve.
    damper->solve();

    // Transfer the damper field.
    damper_field_op.copy();

    // Wave solve.
    local_norm = wave->solve();

    // Collect the l2 norm values from the wave solve to ensure
    // convergence.
    Teuchos::reduceAll<int>( *comm,
                 Teuchos::REDUCE_MAX,
                 int(1),
                 &local_norm,
                &global_norm );

    // Update the iteration count.
    ++num_iter;

    // Barrier before proceeding.
    Teuchos::barrier<int>( *comm );
    }

    // Output results.
    if ( myRank == 0 )
    {
    std::cout << "Iterations to converge: " << num_iter << std::endl;
    std::cout << "L2 norm:                " << global_norm << std::endl;
    }

    return 0;
}
```

The problem was noted to converge for all cases tested in 22 iterations.

8.2 WaveDamper Testing

Using the WaveDamper example, the Data Transfer Kit parallel capabilities have been demonstrated to operate with a variety of hardware and software components. Hardware tested includes shared memory local workstations, the CASL fissile four machines, and a small distributed memory cluster on the University of Wisconsin - Madison campus. Both Intel and GNU compilers have been tested along with both MPICH2 and OpenMPI implementations of the MPI standard. Successful execution of the WaveDamper test was achieved on 120 cores on the small distributed memory cluster.

9 Conclusion

An initial design and implementation of the Data Transfer Kit has been completed. This package attempts to define an interface for data transfer to meet the stated design requirements along with an implementation for parallel operations. This simple interface and implementation should serve as a more general starting point for advanced mesh-based coupling and data structures.

The Data Transfer Kit is currently capable of a simple form of point based mesh coupling. The software infrastructure used in the development of the Data Transfer Kit, instructions on obtaining, building, and testing the Data Transfer Kit, and an outline of how to use the package presented through a concrete example have been provided as an outline for this work. The WaveDamper example has been demonstrated to operate in parallel on various hardware elements from the desktop level to a small scale development cluster, including the CASL fissile four development machines.

References

[1] R. Bartlett, "Teuchos c++ memory management classes, idioms, and related topics: The complete reference (a comprehensive strategy for safe and efficient memory management in c++ for high performance computing)," no. SAND2010-2234, 2010.

[2] C. Farhat, M. Lesoinne, and P. LeTallec, "Load and motion transfer algoritms for fluid/structure interaction problems with non-matching descrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity," *Computational methods in applied mechanics and engineering*, vol. 157, pp. 95–114, 1998.

[3] X. Jiao and M. Heath, "Common-refinement-based data transfer between non-matching meshes in multiphysics simulations," *International Journal for Numerical Methods in Engineering*, vol. 61, pp. 2402–2427, 2004.

[4] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison Wesleyn, 2004.

[5] R. Bartlett, M. Heroux, and J. Willenbring, "Tribits lifecycle model version 1.0: A lean/agile software lifecycle model for research-based computational science and engineering and applied mathematical software," no. SAND2012-0561, 2012.