SAND REPORT

SAND2005-5316
Unlimited Release
August 2005

The Design and Evolution of Tpetra

Paul M. Sexton and Michael A. Heroux
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

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

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

S A N D 2 0 0 5 - 5 3 1 6 xx
Unlimited Release
August 2005

The Design and Evolution of Tpetra

Paul M. Sexton and Michael A. Heroux
Computational Mathematics and Algorithms Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

Tpetra is a package of classes for the construction and use of serial and distributed parallel linear algebra objects. It is one of the base packages in Trilinos. In this document, we discuss the design and implementation of Tpetra. We also discuss the considerations and decisions that led to those designs being chosen.

Acknowledgement

The authors would like to thank the many people who have aided us in the development of Tpetra. In particular, Ross Bartlett, Rob Hoekstra, Heidi Thornquist, Jim Willenbring, and Alan Williams.

Contents

1 Introduction **7**
1.1 Tpetra Dependencies 7
1.2 Scalars and Ordinals 8

2 Linear Algebra Objects **9**
2.1 CompObject 9
2.2 Vector 9
2.3 MultiVector 10
2.4 CisMatrix 10

3 Defining a Distribution **13**
3.1 ElementSpace 13
3.2 Directory 14
3.3 BlockElementSpace 15
3.4 VectorSpace 16

4 Communication **16**
4.1 Platform 17
4.2 Comm 17
4.3 MPI 18

5 Parallel Data Redistribution **19**
5.1 Distributor 19
5.2 Import 20
5.3 Export 21
5.4 DistObject 21

References **23**

Intentionally Left Blank

1 Introduction

Tpetra is an object-oriented C++ library for creating and managing basic linear algebra objects, such as Vectors, MultiVectors, and Sparse Matrices. It implements the Petra Object Model [4]. Tpetra is a direct descendant of Epetra [2], and provides the same capabilities that Epetra does.

Epetra has proven to be very successful. It is robust, portable, and efficient. Its only drawback is that it only works with double-precision real values. It cannot be used with complex or arbitrary-precision data, for example. The goal of Tpetra is to provide the same capabilities as Epetra, and to reuse the same design patterns that have been effective in Epetra, but with the addition of templates. We would like to provide another efficent, robust, and portable linear algebra library that can work with many different types of data.

Tpetra is entirely templated, and this required starting over almost from scratch. Many designs and patterns were reused from Epetra, but all of the code is entirely new. This provided an opportunity to try out some new ideas, something that is not possible with a production library such as Epetra. Some of these turned out to be good ideas, and some of them turned out to be not so good ideas. Some of the good ones have even been incorportated back into Epetra.

We assume that the reader is at least somewhat familiar with the Petra Object Model or with Epetra. In this document we describe how Tpetra is designed, with a focus on how it differs from Epetra. The portions of Tpetra that are implemented identically or nearly identically to the corresponding Epetra parts are not considered very important. They are discussed, but only briefly. What is explained in detail are the parts of Tpetra that work differently than Epetra does, and the parts that function the same, but are implemented differently.

This document contains the majority of the knowledge and wisdom that we have gained over the past few years while writing Tpetra. The descriptions and explanations given here apply to the state of Tpetra at the time of the Trilinos 6.0 release.

1.1 Tpetra Dependencies

Kokkos

Tpetra depends on Kokkos [3] for serial kernels. At present, we are only using one of the Kokkos routines, sparse matrix vector multiplication. Our reason for doing this is to seperate the kernels from the surrounding code, so that more optimized kernels can be developed and used without affecting Tpetra.

Note that the Kokkos routines are purely serial. As such, if we are running in a parallel environment, any redistributions that are needed before or after the computation must be handled by Tpetra.

Teuchos

Tpetra depends on Teuchos [8] for many different things. We use the traits mechanisms in Teuchos::OrdinalTraits and Teuchos::ScalarTraits. We use Teuchos::RefCountPtr [1] throughout Tpetra to manage dynamic storage for us. And we utilize the Teuchos BLAS wrappers for the same reasons that we use Kokkos; it provides a templated interface to the BLAS routines, insulating us from development of more advanced and more optimized BLAS kernels. Finally, we use the Teuchos::CompObject base class to do flops-counting.

Many of these classes, including OrdinalTraits, ScalarTraits, CompObject, and BLAS, were originally part of Tpetra. They were moved to Teuchos because they were found to be useful for other packages. Since the move, active development of these classes has continued, now under the control of the Teuchos team.

C++ Standard Library

Tpetra makes extensive use of the C++ Standard Template Library (STL), which is now incorporated into the C++ Standard Library. We use vectors and maps throughout Tpetra, and call on the STL algorithms quite frequently as well. (Some Tpetra classes use a `Teuchos::Array` in place of a `std::vector` as a data member. This is done mainly so that we have the ability to do array bounds checking. This could be done using vector's `at` member function, but the bounds checking in Teuchos Array can be turned on and off at configure time. This is not possible with the `at` member function.)

1.2 Scalars and Ordinals

A fundamental concept of Tpetra is that of OrdinalType and ScalarType. Tpetra is templated throughout on these two types. (A full discussion of templates is well beyond the scope of this document. Any good C++ book will give you an overview of templates. For an exhaustive reference, see [9].)

The ScalarType is the type of our actual data. In Epetra, the ScalarType is always `double`. In Tpetra, it could be `float`, `double`, `complex<float>`, `complex<double>`, or almost any other type. A user could use a 3x3 dense matrix as a ScalarType if they wanted to. The OrdinalType is primarily an ordering type. For example, we use it as the datatype for element IDs. OrdinalType is also used as a counting type. We use it to store information on how many of something we have. In Epetra, the OrdinalType is always `int`. In Tpetra, it will most likely be an `int` or a `long`. However, it could be any type that is mathematically countable - a type who's values have a one-to-one correspondence to the integers.

Because we don't know which type we're using, it's very important that we don't make the assumption that we can use literals in Tpetra. While `someVar = 5.0` will evaluate correctly if the variable is a `float` or `double`, there's no guarantee that the implicit conversion will succeed if it's an arbitrary-precision object or a vector or matrix.

To solve this problem, we use a design pattern known as traits[1]. We use two Teuchos classes, ScalarTraits and OrdinalTraits. ScalarTraits defines many traits, but most of them, such as machine epsilon, the largest exponent before overflow, etc., are not of interest to us. We are primarily interested in two traits that both ScalarTraits and OrdinalTraits define: one and zero. Zero is the mathematical zero; it is the value such that for all x, $x * 0 = 0$. One is unity, or identity; it is the value such that for all x, $x * 1 = x$.

table of traits values for common types

If a type has these traits defined, and defines the basic operators such as =, +, -, *, /, we can use it as a ScalarType or OrdinalType. Using these, we can do anything we want to. For example, consider this slightly-contrived function that returns the sign of a variable: -1 for negative, +1 for positive, and 0 for zero:

[1] For more information on traits and policies, see Chapter 15 "Traits and Policy Classes" in [9].

```
template <typename OrdinalType>
OrdinalType getSign(OrdinalType const& foo) {
    OrdinalType const zero = Teuchos::OrdinalTraits<OrdinalType>::zero();
    OrdinalType const one = Teuchos::OrdinalTraits<OrdinalType>::one();
    OrdinalType const negOne = zero - one;

    if(foo > zero)
        return(one);
    else if(foo < zero)
        return(negOne);
    else
        return(zero);
}
```

Note that the comparisons are done using equivalence, not equality. This is something that a Tpetra developer must be kept in mind[2].

Also note that we consider an OrdinalType to have a domain of $(-\infty, \infty)$. As a result of this, while it is perfectly legal to use an unsigned type as an OrdinalType, for instance `unsigned int`, it may produce some unexpected results. One such consequence is negative one, which we often use as a placeholder or as an error code in situations where non-negative values are valid. In the case of a 32-bit unsigned int, $-1 = 0 - 1 = 2^{32} - 1$. While odd, this will work fine. It just means that the effective range is now $\left[0, 2^{32} - 2\right]$ instead of $\left[0, 2^{32} - 1\right]$. (We are not concerned with exceeding the domain of an OrdinalType, and we don't check for it. If a user's computations are overflowing, they should switch to a larger type.)

2 Linear Algebra Objects

The main purpose of Tpetra is to manage linear algebra objects. This allows other developers and users to think in terms of vectors, graphs, and matrices without having to concern themselves with how those mathematical concepts are implemented in the machine. As of Trilinos 6.0, Tpetra provides the ability to work with sparse matrices and dense vectors. Vector-vector and matrix-vector operations are supposed.

2.1 CompObject

All Tpetra classes that represent a linear algebra object inherit from `Teuchos::CompObject`. This provides us with the functionality to keep a flops count for that object. The flops count of a CompObject represents the number of floating-point operations that have occured on this image — it is not a global counter. The CompObject class was originally part of Tpetra, and was based off of `Epetra_CompObject`.

2.2 Vector

The most basic unit of linear algebra is the vector. This is implemented in the `Tpetra::Vector` class. It functions in much the same way as `Epetra_Vector`. For proper performance of the BLAS

[2] For the distinction between equivalence and equality, see Item 19 in [6].

routines, it is necessary for the vector elements to be stored in a contiguous array. In Epetra, this is done by using a `double*` array. For storage, we use a `std::vector`. This is possible because the C++ Standard guarantees that a vector will store its data in contiguous form[3].

Vector can be used to perform many mathematical operations, such as scaling, norms, dot products, and elementwise multiplies. But it is perhaps more useful in conjunction with another Tpetra class, CisMatrix.

2.3 MultiVector

Another very useful concept is a MultiVector. A MultiVector is a collection of Vectors that all have the same dimensions. It can be thought of as a generalization of a dense matrix. Epetra supports for using a MultiVector in many of the situations where a Vector can be used. In this way, rudimentary matrix-matrix operations are provided.

As of this writing, Tpetra does not yet have a MultiVector class. But development of one is planned, and it will have all the capabilities of an Epetra MultiVector, and a very similar interface. In Epetra, Vector is implemented as a specialization of MultiVectorm, and Tpetra will most likely have that same implementation.

Future Work: An idea we will be experimenting with in Tpetra is making the vectors in a MultiVector independent entities, with regard to allocation and deallocation. A MultiVector and a Vector will be able to share data, for instance. For efficiency reasons, we still need to allocate all of the vectors in a MultiVector contiguously. We will maintain a seperate reference count for each column in the MultiVector, and deallocate the array only when all of the reference counts have reached zero.

2.4 CisMatrix

CisMatrix is a generalization of the compressed-row sparse matrix found in Epetra_CrsMatrix. A Tpetra::CisMatrix can be either row-oriented or column-oriented. This property is set at construction.

Matrix Distributions

Due to this generalization, CisMatrix does not deal just with the row distribution and the column distribution. Instead, we use the concept of the primary distribution and the secondary distribution. In a row-oriented matrix, the primary distribution is the row distribution. In a column-oriented matrix, the primary distribution is the column distribution. Almost all of CisMatrix deals with the primary and secondary distributions instead of dealing with the row and column distributions directly. This is what allows us to support both orientations with the same code.

Consider the mathematical equation for a matrix vector multiply: $Ax = y$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \tag{1}$$

[3] See Section 6.2.3 in [5] and Item 16 in [6].

Add captions to this

The row distribution contains an entry for each row, and every image that has an non-zero in that row owns that entry. The column distribution works the same way. In a row-oriented matrix, each entry in the row distribution will (usually) be uniquely owned, and entries in the column distribution may be multiply owned. In a column-oriented matrix, the reverse is true.

The domain distribution specifies the distribution of an x vector that we are prepared to accept. The range distribution specifies the distribution of a y vector that we are prepared to accept. (Note that while they describe the x and y vectors, the domain and range distributions are properties of the matrix, not of the vectors.)

FillComplete

A CisMatrix object exists in two different phase during its lifetime. When it is first created, the user will be submitting entries. The matrix is in flux, and no computations can be done. When the user is finished entering data, they signal this by calling `fillComplete`. After this is done, the matrix enters the second phase. The matrix is now considered to be static, and no entries can be modified. But it can now be used to do sparse matrix-vector multiplications, and sparse matrix-vector solves. The call to fillComplete is a pivotal moment, and the function accomplishes several things.

1. The domain and range distributions, which were previously undefined, are now set. If the user did not specify them, they will both be set to the primary distribution. (Note that this is only possible with a square matrix. With a rectangular matrix, the user must specify them. There are no restriction on how the domain and range distributions are set up, provided that their global lengths match the global lengths of the matrix dimensions. For example, a domain distribution where all elements were owned by Image 0 is perfectly legal, but performance will suffer due to the additional communication that may be needed.

   It might seem more logical to have a default behavior that sets the domain equal to the column distribution, and the range equal to the row distribution. This would work with both square and rectangular matrices. But it's not possible, due to the way the functions are set up. The fillComplete function with no arguments (the default case) would have to call getRowDistribution() and getColumnDistribution(), and uses the reference returned as the parameters to the fillComplete function with two arguments (the non-default case). But the secondary distribution might not be defined yet, and trying to access it would throw an exception.

2. The matrix data is converted into an optimized form for use with the Kokkos kernels. Prior to fillComplete, data is stored in a map of maps. After fillComplete, data is stored in Classical Harwell-Boeing form, using three std::vector objects to store the pntr, indx, and values arrays.

   Diagram showing pre- and post-fillComplete layouts (from thesis).

3. If the user did not specify a secondary distribution at construction, it will be generated now.

4. The domain, range, row, and column distributions will be analyzed for compatibility. If necessary, we will construct Import and Export objects to convert between them.

Of these three tasks, generating the secondary distribution is the most difficult. (Converting the data to a contiguous form involves straightforward copying, and setting the domain and range distributions are simple assignments.) It is fairly likely that a single column will contain entries from multiple rows. This means that those elements will be multiply owned, and so to construct the

ElementSpace object that we will base our VectorSpace on, we need to know which columns we own.

We accomplish this by calling several STL algorithms in sequence. First, we make a copy of our newly-created indx array, which contains the secondary indices of every non-zero we own. We call `sort` on this vector, which will sort the vector in place. Then we call `unique` on the sorted vector. This removes any duplicate entries. And because `unique` is a remove-like algorithm, we have to follow it with a call to `std::vector::erase`[4]. We are now left with a vector containing all the columns we have entries in, sorted in ascending order. This is what we will pass into the ElementSpace constructor as the myGIDs parameter. (We will use the size of the vector for numMyElements, and set numGlobalElements to -1 to have ElementSpace compute that for us.)

Once fillComplete has been called, subsequent calls should be a no-op and should not generate a warning or error.

Apply

The one exception to this orientation-agnostic policy is when we are doing a matrix-vector multiplication, implemented in the `apply` method. This is because regardless of the orientation, the mathematical matrix represented by the class is the same. In either case, the x vector will need to match the matrix's column dimensions, and the y vector will need to match the matrix's row dimensions. (If we are doing an apply on the transpose of the matrix, these matchups are reversed.)

There are four possible cases for an apply, but we can get by with having only two routines. The matrix may be row-oriented or column-oriented, and we may or may not be doing a transpose. Because a Kokkos::CisMatrix can have either orientation, our apply function does not need to take that into account. All we need to worry about is whether or not we are doing a transpose, which is slightly more complex than a non-transpose apply.

Our basic strategy is the same as that in Epetra: We have an x vector that matches a domain distribution, and a y vector that matches a range distribution. If the domain does not match the column distribution, we need to do an Import. If the row distribution does not match the range, we need to do an Export. During the call to fillComplete, we initialized a Tpetra::Import object and a Tpetra::Export object if they were needed. (The domain and range distributions are fixed during that function, so we can safely compare them to the row and column distributions.) The Importer will import from the domain distribution to the column distribution. The Exporter will export from the row distribution to the range distribution.

$$domain \rightarrow column \qquad row \rightarrow range$$

This works very well for a "normal" matrix-vector multiplication. Our importer will convert from the domain distribution to the column distribution, and our exporter will convert from the row distribution to the range distribution.

Doing a transpose multiplication is trickier. In that case, x and y will correspond to the domain and range distributions of A^T. These are equivalent to the range and domain distributions of A, respectively. To do the actual multiplication, we need x to match the column distribution of A^T, and we need y to match the row distribution of A^T. Again, these are equivalent to the row and column distributions of A, respectively. Our situation now looks like this:

[4] See Item 32 in [6] for a full explanation

$$range \rightarrow row \qquad column \rightarrow domain$$

(Remember that we can only deal with the distributions of A, as A^T never really exists, and certainly never has any distributions defined for it.) We don't have any importers or exporters to handle these cases. But we can use the importer and exporter we do have to handle these cases by doing a reverse import and a reverse export. By doing an import using our exporter, we can import from the range distribution to the row distribution. And by doing an export using our importer, we can export from the column distribution to the domain distribution.

3 Defining a Distribution

In order to use our linear algebra objects on a parallel machine, it is necessary to specify how the data is distributed. Defining these distributions forms one of the foundations of Tpetra. We accomplish this by using several classes in conjunction.

Epetra Migration Note: Tpetra's ElementSpace and BlockElementSpace are functionally equivalent to Epetra's Map and BlockMap classes, respectively. However, in Epetra, Map *isA* BlockMap, and is implemented as a BlockMap with a single point per block. In Tpetra, the relationship is reversed. ElementSpace is a stand-alone class, and BlockElementSpace *hasA* ElementSpace. BlockElementSpace is implemented by storing the number of points for each element in the ElementSpace.

Our rationale for this change is that a block distribution involves storing additional data, and creates an additional overhead. Even though most applications will not need it, Epetra imposes that extra overhead on all users by making Map inherit from BlockMap. In Tpetra, only the applications that need a block distribution will use a BlockElementSpace, and only they will bear the additional burden.

The functionality found in Epetra_LocalMap has been integrated into ElementSpace. An ElementSpace object will automatically determine if the distribution the user has specified to it is a global or local (replicated) distribution.

3.1 ElementSpace

ElementSpace defines a parallel distribution of data. The atomic unit for an ElementSpace is an Element. Given an array of Elements, ElementSpace will assign a GID and an LID to each Element. Code that uses an ElementSpace can query it to find out what elements they own, as well as finding out who owns a specified GID.

There are three ElementSpace constructors:

1. **Uniform Contiguous Distribution** The first constructor will create a Tpetra-defined contiguous distribution. The caller specifies how many global elements there are, and ElementSpace will assign GIDs to each element, and distribute them to one of the images. Global IDs will be in the range [indexBase, indexBase + numGlobalElements). Each image will be given (numGlobalElements / numImages) elements. If there are any leftover elements, one additional element will be given to each of the first (numGlobalElements % numImages) images. Each

element is guaranteed to be uniquely-owned. (In other words, for every GID existing in the ElementSpace, exactly one image will own it.)

2. **User-Defined Contiguous Distribution** The second constructor will create a user-defined contiguous distribution. The caller specifies how many global elements there are, and how many are owned by this image. ElementSpace will assign GIDs to each element, and attempt to distribute them the way the caller requested. If the global sum of numMyElements does not equal numGlobalElements on any image, an exception will be thrown. (The caller can have ElementSpace compute numGlobalElements by passing -1 as that parameter.)

   As with the uniform contiguous distribution, Global IDs will still be given out in the range [indexBase, indexBase + numGlobalElements), and each element is still guaranteed to be uniquely-owned.

3. **Non-Contiguous Distribution** The third constructor will create a user-defined non-contiguous distribution. The caller specifies how many global elements there are, how many are owned by this image, and an STL vector containing the GIDs owned by this image. The myGIDs array does not need to be in a sorted order. However, LIDs will be assigned in order - myGIDs[i] will have LID i.

   Unlike the first two constructors, there are almost no restrictions on the GIDs. A GID can be owned by one or more images, and can have any value, provided it is not less that indexBase.

Note that the distributions generated by the different constructors are not mutually exclusive. Any distribution that can be created in the second constructor can also be created in the third constructor, and any distribution that can be created in the first constructor can also be created in either the second or third constructor.

ElementSpace is a static class, meaning that after construction, it will not change state at all. All of the member functions return information; none of them change class data members. Thus, it is possible to have a ElementSpace object declared as const, since there are no non-const member functions (with the exception of the assignment operator.)

Future Work: We would like to provide the user with the ability to find out if any of the elements in their ElementSpace object are multiply-owned, and if so, which. This will involve refactoring both ElementSpace and Directory, but it should not be too much work. This will provide additional functionality to Tpetra, as well as to the user. For example, in Tpetra::Import, we could throw an exception if the caller specified a source distribution that was multiply-owned.

A difference from Epetra is the way ElementSpace stores the LIDs and GIDs. They are stored using a pair of STL maps. In lgMap_, the LID is the key and the GID is the value. In glMap_, the GID is the key and the LID is the value.

3.2 Directory

Directory is a class that is never used directly by the user. It is an auxiliary class that is used by ElementSpace. (In fact, the getRemoteIDList functions in ElementSpace do nothing but call Directory::getDirectoryEntries, and pass the results back up.)

ElementSpace only stores information about the elements it owns, and about the global size of the distribution. It knows nothing at all about the elements owned by other images. This information is often needed by classes utilizing an ElementSpace, and Directory keeps track of it. Given a list

of GIDs, the Directory will tell you which images own those elements, and (optionally) the LIDs on those images that correspond to the GIDs.

Because inter-node communication is so expensive, it would be wasteful and possibly prohibitavely slow to query the other images every time a request was made. Fortunately, because an ElementSpace is static after construction, we don't have to. When a Directory is constructed, it will communicate with itself (using a Distributor). When this is finished, every image will know the ImageID and LID for every element. When the user calls getDirectoryEntries, we simply look up those elements in our local database.

3.3 BlockElementSpace

BlockElementSpace builds on top of ElementSpace. Given a global distribution of elements, it defines the distribution of points within each element. There are two types of BlockElementSpace objects that can be constructed. One has a constant element size, and one has variable element sizes.

BlockElementSpace is functionally equivalent to Epetra::BlockMap, and its internal structure and implementation is nearly identical. The one new concept that does not exist in Epetra is that of compatibility. Compatibility was not an original aim, rather, it is a result of Tpetra having a VectorSpace class. (This will be explained more thoroughly in the section on VectorSpace.)

Compatibility

A BlockElementSpace can generate a new ElementSpace object, known as a "compatible ElementSpace", using the member function `generateCompatibleElementSpace()`. This new ElementSpace will be templated on the same OrdinalType as the BlockElementSpace that created it.

A "compatible" ElementSpace is defined to be an ElementSpace where there is a 1 to 1 correspondance between the elements of the compatible ES and the points of the BlockElementSpace that generated it. (Although ES has no concept of points, if it were viewed as a BES with a constant element size of one (i.e. each block has 1 point), then it should have the same number of points as the generating BES, both globally and locally on every image.)

The real meaning of compatibility is that they can be used to construct VectorSpaces that are compatible. This means that vectors created using those VectorSpace objects will have the same lengths, and so can be used together in vector-vector operations. It is guaranteed that VectorSpace objects constructed using compatible ES/BES objects will be compatible themselves, as defined by VectorSpace's `isCompatible` member function.

Add diagram (lab notebook p.17)?

There are two main benefits to using compatible ElementSpaces. One is that if you are trying to determine if a BES and an ES will produce compatible VectorSpaces (and therefore, compatible Vectors), instead of having to try and compare them directly, you can simply generate a compatible ElementSpace from the BlockElementSpace. You can then compare the two ElementSpace objects using the == operator or the isSameAs method in ElementSpace.

The second, and much more useful, benefit is this: By creating a compatible ES and using that to generate Vectors, we have given globally-accesible IDs to the data represented by BES points. But

because VectorSpace keeps the original BES passed in, that data will still only be imported and exported as blocks.

3.4 VectorSpace

Although distributions are created using ElementSpace and BlockElementSpace, VectorSpace is the "base" distribution class for the linear algebra objects such as Vector, MultiVector, and CisMatrix. VectorSpace provides an insulating layer against whether an ElementSpace or BlockElementSpace is being used. From the point of view of a Tpetra::Vector, for example, there is just a VectorSpace, specifying how many entries are in the global vector, and how many of those entries belong to us.

If a VectorSpace is built on an ElementSpace, there is a one-to-one correspondence between elements in the ElementSpace, and entries in the VectorSpace. If a VectorSpace is built on a BlockElementSpace, there is a one-to-one correspondence between points in the BlockElementSpace, and entries in the VectorSpace. This is done by having the BlockElementSpace create a "compatible" ElementSpace, where each point in the BlockElementSpace corresponds to an element in the compatible ElementSpace.

Important Note: Points in a BlockElementSpace do not have any global identifiers, and are never supposed to be redistributed individually. The whole point of having a BlockElementSpace is so that the data represented by the points will always be moved around in blocks, and not individually. But by generating a compatible ElementSpace, and building a VectorSpace on top of that compatible ElementSpace, we have effectively given a global identifer to each point, since each entry in a VectorSpace has both a local index and a global index. Because of this, it is very important that we redistribute data based on the original ElementSpace, and not based on the compatible ElementSpace.

To accomplish this, it is necessary for the DistObject to be aware of how its VectorSpace was constructed. In addition, it is necessary for the DistObject to be able to query the BlockElementSpace object. This is because it is not enough for the DistObject to know that some of its entries need to be redistributed as blocks. It needs to know which entries are part of which blocks.

This information is only needed in the functions inherited from DistObject. The rest of the class should not depend on knowing about this, and should be written to work with a "generic" VectorSpace.

4 Communication

Parallel communication is done using Tpetra::Comm, the Tpetra Communicator class. This is the same as is done in Epetra. Comm provides an insulating layer between the actual communications library being used and the rest of Tpetra. Whether we are using serial, MPI, shared memory, or some other setup is only relevant to the Comm class.

The most striking difference between Epetra and Tpetra with regard to communication is Processors vs. Images. In Epetra, nodes are referred to by their Processor ID (PID), and we keep track of the number of Processors in the communicator. In Tpetra, we have switched to a different nomenclature with nearly identical semantics. We recognize that a single processor may be running multiple MPI jobs, and that conversely, several processors may be running a single MPI job. A "memory image" is our concept for a virtual node. There is exactly one image for each copy of

Tpetra code that is running concurrently. So instead of `myPID` and `numProcs`, we have `myImageID` and `numImages`. (ImageID should always be written out; it is not acceptable to abbreviate it IID.)

4.1 Platform

In Epetra, Comm is responsible for collective operations, managing PIDs, and creating Directory and Distributor instances. In Tpetra, we decided to split this up. We wrote a new class, Platform, to handle some of these tasks. Initially, Platform was responsible for managing ImageIDs, creating Comm instances, creating Distributor instances, and creating Directory instances. Recent refactoring has removed several of its duties, and now it is only responsible for creating Comm instances. (These changes will be explained in later sections.)

We wanted to provide the same interface as Epetra, and use that same system of an abstract base class to hide whether we were using serial, MPI, etc., from the surrounding code. We also wanted to add templated support. The most logical way to do this would be to template the individual member functions. Then a user could have a single Comm object, but would be able to call functions like broadcast, sumAll, etc. with different types of data. Unfortunately, that is simply not possible[5]. Virtual member functions cannot be templated. We can template them at the class level, but not at the function level.

We spent over a month puzzling over this problem. There seemed to simply be no solution. C++ had us backed into a corner. So we compromised, and templated Platform and Comm at the class level. This was a working solution, but it was awkward. The user almost always has to create two Platform objects - one templated on OrdinalType, ScalarType, and one templated on OrdinalType, OrdinalType. The OT, ST Platform is needed to create objects like VectorSpace and Vector that have a ScalarType defined. But we can't pass that Platform into a class like ElementSpace, because ElementSpace is templated only on the OrdinalType, not on the ScalarType. We definately didn't want to template ElementSpace on the ScalarType, because we want to be able to reuse a distribution with differently-typed VectorSpaces.

Future Work: In the summer of 2005, we came up with a new approach. The problem was that virtual member functions and templates are mutually exclusive. Why not get rid of the inheritance and keep the templates? This was coded up in a new Tpetra class named OmniPlatform. The class is not templated, but the member functions are. We could pass a single Platform instance into every class, and allow those classes to have the OmniPlatform generate typed Comm instances for them. (This is only possible because Platform no longer has any role except to create Comms.) In place of the inheritance, we use preprocessor macros to include and exclude the routines for creating a SerialPlatform, a MpiPlatform, etc.

Initial testing shows that OmniPlatform seems to work very well. But preprocessor macros are ugly and brittle, and we were reluctant to depend on them for such a vital part of Tpetra. The version of Tpetra in Trilinos 6.0 does not use OmniPlatform. But we may switch to it in the future.

4.2 Comm

Tpetra::Comm provides the same collective communication operations that Epetra::Comm does: barrier, broadcast, gather all, global sum, global min, global max, and scanSum (parallel prefix sum). In addition, it provides several point-to-point operations that while present in Epetra, they

[5] See Section 8.1.1, page 98, in [9]

were not part of Comm. The new operations are sumAllAndScatter, blocking send, blocking receive, and the doPostsAndWaits family of functions.

These new functions are the result of a design decision made by the Tpetra team in July 2005. In Epetra, Distributor is a communications class. There is a Distributor abstract base class, and SerialDistributor and MpiDistributor implementations. This means that MPI code exists both in Epetra::MpiComm and in Epetra::MpiDistributor. In Tpetra, all MPI interaction is done through the Comm class. Tpetra::Distributor is just another Tpetra class that depends on Comm for communication.

Epetra::MpiDistributor did most of the MPI calls in the doPosts and doWaits functions. But some communication was done in the setup functions. The blocking send, blocking receive, and sumAllAndScatter functions were added to Tpetra::Comm so that these setup functions in Distributor could be implementation-independent.

The doPosts and doWaits functions in Epetra::MpiDistributor perform specialized and somewhat complicated operations that do not need to be generally available. This is one reason why they were moved entirely into Comm, instead of having wrappers added for the specific MPI calls they made.

Distributor now functions much like Import and Export do. They do the initial setup, but when the actual communication is done, it is done by a different class, with the setup class given as a parameter. In the case of Import and Export, the actual communication is done by the DistObject class, with an Import or Export object passed into the doImport or doExport member function of DistObject. In the case of Distributor, it is passed as a parameter to the doPosts, doWaits, doPostsAndWaits, doReversePosts, doReverseWaits, and doReversePostsAndWaits member functions of Comm.

Epetra Migration Note: The Epetra Distributor functions `do` and `doReverse` have been renamed `doPostsAndWaits` and `doReversePostsAndWaits`, respectively. Their functionality is unchanged.

4.3 MPI

In Epetra, dealing with MPI was straightforward. Ints and doubles were passed directly, using the MPI_INT and MPI_DOUBLE datatypes. When a DistObject was transmitted, it was packed into a buffer and treated as a char* array, using the MPI_CHAR datatype. In Tpetra, it's not that simple. When doing a simple collective operation, such as sumAll, we don't know what datatype we're using, since the Comm class is templated. And when communicating a DistObject, we can't assume anything about the memory layout of the ScalarType or OrdinalType we're using. It may contain pointers to heap arrays, or any number of things.

Once again, our solution involves traits. The Tpetra::MpiTraits class works very similarly to OrdinalTraits and ScalarTraits. It provides four traits: datatype, count, sumOp, maxOp, and minOp.

datatype This returns a variable of type MPI_Datatype. This is the datatype that MPI thinks it is receiving. For example, MPI_BYTE, MPI_FLOAT, or MPI_INT.

count This returns a variable of type int. This is the number of objects that MPI thinks it is receiving. This is a multiple of userCount, a parameter passed to the count function. userCount is the number of objects that the caller thinks it is sending to Tpetra::Comm.

sumOp This returns a variable of type MPI_Op. For the built-in datatypes, this is MPI_SUM. But for non-built in datatypes, such as complex<T>, or a matrix or vector, we need to define our own sum operation, and this trait is how we pass that into MPI.

maxOp This functions the same way that sumOp does, except that it returns the operation for finding the global maximum value. For complex<T>, we compare the objects by their magnitude, since there is no such thing as less than or greater than for a complex value. We still need to return a complex<T> variable, so we return the first one we encountered that had the largest magnitude.

minOp This functions the same way that maxOp does (including its handling of complex values), except that it returns the operation for finding the global minimum value.

Defining the datatype and count traits for a new datatype are quite straightforward. Defining the sumOp, maxOp, and minOp traits are fairly involved, since they involve defining new functions for MPI to use. The traits defined for complex<T> should provide a decent template to copy. If that is not enough, refer to [7].

Warning: The way we handle complex values is to tell MPI we're passing it two values of type T for each value of type complex<T> the user passes us. This works quite well, and allows us to not have to define a new datatype, since MPI already supports arrays. However, this assumes that a complex variable's memory layout consists only of the two T variables. (In other words: This assumes that if we take the address of a complex<T> variable, cast it to a T* pointer, and pass that pointer into MPI, MPI will find a T variable there. Furthermore, incrementing that pointer must produce a seconed T variable.)

This has been the case on every machine the author has tested it on, and it is possible that such a layout is required by the C++ Standard. However, as of this writing, we do not know that for sure. So it is possible that there is a machine on which this will break. In which case we will just have to declare a custom datatype, which can be handled by the existing datatype trait.

It does not matter in which order the real and imaginary components occur, as long as all nodes use that same layout.

5 Parallel Data Redistribution

Redistributing data in a parallel environment is a multi-step process, involving several classes. We start out with a class that derives from the DistObject class. We define an Import or Export object, giving it the distribution we have now, and the distribution we want to end up with, as constructor parameters. We then call doImport or doExport on our DistObject. That is as far as the user sees it. There are still several more steps that take place inside that call to doImport or doExport.

5.1 Distributor

Distributor is the workhorse of data redistribution. It is responsible for determining what to send, what to receive, and for actually doing to communication.

There are two kinds of Distributor objects that the user can create. In one, we know what we want to receive, and we need to find out what we have to send to others. In the second, we know what

we want to send, and we need to find out what we will be receiving from others. (These correspond to import and export operations, respectively.) This is not done by having two constructors. Instead, there is a single constructor that doesn't really do anything. Later, the user calls either `createFromSends` or `createFromRecvs`. This are the functions that really initialize the object.

Distributor::createFromSends is the simpler case. We tell the Distributor how many elements we're exporting (numExportIDs), and which images to send those elements to (exportImageIDs). It tells us how many elements we'll be receiving (numRemoteIDs). Distributor::createFromRecvs is a more complicated case, since the Distributor has to determine what each image has to send for us to receive what we're asking for.

After createFromSends or createFromRecvs has been called, the Distributor object is essentially static. It can be used to do multiple (identical) redistributions with no additional cost. This actual communication is done by calling doPosts, which sends all the data out, and by then calling doWaits, which will wait for all the data to show up (which may take awhile, since each image has to send data to multiple images.) The reason doPosts and doWaits are seperate functions is so we can do local computations in the interim that we would otherwise spend idle. If we are not concerned about this, or do not have any work we could be doing, we can call doPostsAndWaits, which combines the two for us.

Epetra Migration Note: In Epetra, the Distributor class is responsible for the entire process. This is the way Tpetra's Distributor started as well. In the summer of 2005, Tpetra's Distributor was refactored. The doPosts, doWaits, doPostsAndWaits, doReversePosts, doReverseWaits, and doReversePostsAndWaits functions that used to be in Distributor are now in Comm.

Add sample code here (old way vs. new way)

5.2 Import

Import is another "static" class. After construction, it cannot be modified, and is used solely to provide information to other classes, such as Distributor. Import defines a mapping between two ElementSpace objects: a Source ElementSpace that defines the distribution we have now, and a Target ElementSpace that defines the distribution that we want to end up with. As its name suggests, it is used in situations where we want elements that are currently owned by other images. (For example, replicating the x vector in a row-wise matrix-vector multiplication.) Import does not do any work itself; much like ElementSpace, it exists to provide information to other classes.

The Elements owned by this image in the Target distribution are divided into three categories:

same These elements are identical between the source and target. A given GID has the same LID in both distributions.

permute These elements are owned by this image in both source and target, but they are in a permuted order. The LIDs don't match.

remote These elements are owned by this image in the target distribution, but are owned by a different image in the source distribution.

Note that same is restricted to the first contiguous range of same elements. If GIDs 0,1, and 2 have same LIDs, GIDs 3 and 4 have permuted LIDs, and GIDs 5 and 6 have same LIDs, GIDs 5 and 6 will be considered permuted, not same. There is very little performance hit, if any, as a result of

this. This is because both same and permute IDs will be copied locally, and a local memory copy is very inexpensive, relative to the cost of importing the remote elements from off-image.

Having computed this, the Import instances on each image will communicate with each other. Using a Distributor, they will tell each othert what GIDs they are expecting to receive, and will learn which GIDs they need to send out, and who to send them to.

Figure with tri-diagonal import example

5.3 Export

Export is very similar to Import. At first glance, they may seem identical, or nearly so. Do not be fooled. The differences between them, although subtle, have important ramifications. Whereas Import is used when we need to replicate some data, Export is used when we need tto get rid of some data. (For example, combining the multiple y vectors resulting from a column-wise matrix-vector multiplication.)

The Elements owned by this image in the Source distribution are divided into three categories: same, permute, and export. Same and permute have the same meaning as in Import. But instead of making a list of elements owned by target that are not owned by source, we make a list of elements owned by source that are not owned by target. These are the elements that we will be exporting to other images.

After computing this, the Export instances communicate via a Distributor. We already know the GIDs that target owns - that's already in the ElementSpace. What we learn from this is which of those elements are only being locally copied, which of them are only being imported (but not copied locally), and which elements we are receiving from multiple sources, and need to combine.

Figure with tri-diagonal export example

Import/Export Notes

Even though most linear algebra objects use VectorSpace to define distributions, Import and Export take in ElementSpaces, not VectorSpaces. This is because some objects (like a Graph) do not have a VectorSpace associated with them. So Import and Export should only deal with ElementSpaces. We don't need to deal with BlockElementSpace, because in the context of an import or export, we don't care about points. That's for the implementing DistObject to worry about. Here we are only concerned with elements.

It is possible that the user could try and create an Import or Export from mismatched ElementSpaces. For now we are not going to worry about that possibility, and will wait to see if that becomes a problem.

5.4 DistObject

DistObject is a base class that all Tpetra objects that can redistribute their data derive from. DistObject defines the doImport and doExport functions that the user will call to execute a redistribution. These functions are not overridden by the derived class.

DistObject also defines four pure virtual functions that the derived class must implement. These contain the class-specific instructions for packing and unpacking the data in that class that is associated with an Element. (For example, in a Vector this is just the value. In a MultiVector or CisMatrix, this is the secondary index and the value.)

- checkSizes This function will check that the source DistObject passed in can be used with the `this` object. This usually invoves doing a `dynamic_cast` to test that both objects are of the same class (e.g. Vector).
- copyAndPermute This function will copy the same and permute entries from the souce object into the this object.
- packAndPrepare This function will take the elements we are exporting and pack them up into a buffer.
- unpackAndCombine This function will take in a buffer of packed elements, and unpack them. If any of the imported elements already exist, we will use the CombineMode specified to combine them together.

References

[1] Roscoe A. Bartlett. Teuchos::RefCountPtr Beginner's Guide. Technical Report SAND2004-3268, Sandia National Laboratories, 2004.

[2] Michael A. Heroux. Epetra home page. http://software.sandia.gov/Trilinos/packages/epetra, 2005.

[3] Michael A. Heroux. Kokkos home page. http://software.sandia.gov/Trilinos/packages/kokkos, 2005.

[4] Michael A. Heroux, Robert J. Hoekstra, and Alan Williams. The Petra Object Model for Parallel Linear Algebra Computations. Technical Report SAND2005-5316xxx, Sandia National Laboratories, 2005.

[5] Nicolai M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, 1999.

[6] Scott Meyers. *Effective STL*. Addison-Wesley, 2001.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1, The MPI core*. The MIT Press, 1998.

[8] Heidi Thornquist. Teuchos home page. http://software.sandia.gov/Trilinos/packages/teuchos, 2005.

[9] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates The Complete Guide*. Addison-Wesley, 2003.