Design Document for the MÜLOO Project

Jérémie Gaidamour[1], Jonathan J. Hu[1], Christopher M. Siefert[2], and Raymond S. Tuminaro[1]

[1]Scalable Algorithms Department
[2]Computational Shock & Multiphysics Department

September 16, 2010

Contents

1 Introduction

This planning document is intended to record some design ideas and decisions for the new algebraic multigrid libary MÜLOO. MÜLOO is meant to be both a research vehicle and a replacement for ML in existing applications. (Obviously, those codes would have to use TPETRA instead of EPETRA.) Hence, MÜLOO must be flexible enough to support new algorithm development, but also robust enough for everyday use. This means that MÜLOO *must* contain the key algorithms in ML that are in use today: standard smoothed aggregation, Petrov Galerkin for nonsymmetric systems, and reformulated Maxwell.

This document is organized as follows. In §2, we discuss the Matlab prototype multigrid code MÜMAT. In §3, we adresses the design issues which are specific to MÜLOO.

2 MÜMAT Matlab Project

An important part of the MÜLOO project is the matlab code called MÜMAT. MÜMAT serves many purposes. First, it is intended to guide the design of MÜLOO by allowing for rapid prototyping and testing of algorithms and design decisions. Second, it is intended to be the first introduction to MÜLOO for students and new developers. Third, it acts as a living part of the design documentation.

A main tenet is that MÜMAT will mirror the design of MÜLOO. Any capability in MÜLOO should be present in MÜMAT, but not necessarily vice versa. While MÜMAT is written in Matlab, it makes heavy use of classes and inheritance.

2.1 Design of the `operator` Class

In this section, we cover the main design issues for the MÜMAT `operator` class. An `operator` is simply a matrix whose data is either owned by an application or by MÜMAT. The data may be stored in point or block form, but this is secondary to how the `operator` will be accessed. Accessibility is arguably the more important consideration because all phases of setup and solve require `operator` operations. Access requirements may change from one phase to the next, due to algorithmic reasons. However, using a view different than the native storage format may incur a significant performance penalty.

2.1.1 When One View Just Isn't Enough

The most common format for an `operator` will almost certainly be the point compressed row storage format, also known as CRS. Some matrices may be stored in a block format, but this will probably be much rarer.

Once an `operator` is constructed, however, we want the ability to access it as if it were stored in some format other than its native storage format. There appear to be two main use cases:

1. The matrix is stored natively as a point CRS matrix. Point and block `getrow` access are required.

2. The matrix is stored natively as a block matrix. Either point access[1] or block

[1] This is an expensive operation. Blocks are likely stored as small dense matrices or as the fundamental scalar type. In either case, this involves lots of indirect lookups and copies.

access with a different size block is required [2].

These could occur, for example, in smoothers like Gauss-Seidel that need `getrow` access, in aggregation during the setup of the prolongator, or in a matrix-matrix multiply. Figure 1 summarizes the combination of native format and view possibilities.

Native format	View		Insert	
	block	point	block	point
block	Y†	Y	Y†	?
point	Y	Y	N	Y

†Includes different blocking than native format

Figure 1: Operator row access feature matrix.

Here are a couple comments on matrix views.

Point Views: This is the easier case, albeit more expensive. For a matrix stored in a block format, we wish to extract a point row. This requires determining the block row that contains the point row. Then it is just a matter of extracting the correct point row.

Block Views: First suppose the matrix is natively in point format. If the matrix is thought instead to be of constant block size, then this is a simple lookup of the correct point rows and translation into dense block entries.

2.1.2 Multiple Diagonals

For both relaxation and prolongator smoothing, we may want to experiment with different types of diagonals. An operator could have associated with it more than one point diagonal (e.g., the true point diagonal and a lumped version) and more than one block diagonal (differently sized blocks, same sized blocks but different values). Furthermore, the blocks in the diagonal might be distributed, e.g., if we are solving a blocked system

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

[2]This could be expensive if the requested block size is larger than but not evenly divisible by the smaller native size, or if the requested block size is smaller than the native size.

a block diagonal entry d_{ii} might consist of point entries from all four blocks:

$$d_{ii} = \begin{bmatrix} a_{ii} & b_{ii} \\ c_{ii} & d_{ii} \end{bmatrix}.$$

We may also want to play with overlapped block diagonals, such as for Arnold-Falk-Winther smoothing.

Questions

1. The most reasonable place for the diagonals to be stored is with the operator. How do we indicate that we want to use a particular one? How do we check to see whether a particular diagonal is present?

2. Should we be able to register a new diagonal at anytime? If so, how do we go about this? Or, should we only be able to construct the diagonals when the operator is built?

3. When we define a diagonal, this fixes the row map. Does there have to be a `getrow` that corresponds to this map?

2.1.3 Other Considerations

We may also want to be able to do a deep copy of a matrix from one storage type to another. This means that we'll need converters from point to block and vice-versa.

2.1.4 An `operator` strawman

1. The `operator` must have pairs of `getrow`s and diagonals. It is not possible to have a `getrow` without a corresponding diagonal, or vice-versa.

2. The `operator` has a "state" manager that remembers the current view of the matrix and is the only way to change the view.

Comment: One design possibility is to *always* store the matrix as a CSR matrix, and only use views to access it as a block matrix. This means giving up performance, but simplifies our interactions with 3rd party linear algebra packages.

2.2 Current design of `Operator` related classes

Figure 2 presents the current design of the `Operator` class. Note that only the classes `Operator` and `Diagonal` are directely used on the other parts of MÜMAT. SetXXX() methods are not represented on this diagram. The `Map` class is also missing.

2.2.1 The `Operator` class

An `Operator` is simply a matrix. In MÜMAT, the matrix data are natively stored as a MATLAB matrix (encapsulated in the `Operator` object) and you can access directly to internal data storage by using the method GetMatrixData(). This accessor is used in MÜMAT for convenience and efficiency reasons.

Some MÜMAT algorithms mimic the fact that we only have such row accessors: These algorithms loops over the rows and blocks and extract blocks from the MATLAB matrix:

2.2.2 The view mecanism

One goal of the `Operator` class is to be a thin interface layer between Epetra/Tpetra and MÜLOO (to allow us to be compatible with both libraries). An other important goal is the ability to access matrix as if it were stored in some format other than its native storage format. For example, we would like to be able to access block matrices as if they were stored in a point entry storage formats or with another block size. Each matrix format is called a `View` and the defaut view of an `Operator` is the native data format of the matrix. Additional "virtual" views can be defined to describe the way we want to access the data of an operator. The views of an `Operator` are stored in a `ViewTable` which is just a list of `View`. Each view is fully described by two Map objects (for row and columns).

Remark 1. *At the moment, if we want to do an operation between two Operator (let's say OpA+OpB) with different internal data storage, we cannot. There is no mecanism to do that. The easiest way to perform such operation is to use the point format as the common language between operator (slow). A better way is to find the "greatest common block format". Maybe we can use a method to compare Maps:*

```
function Operator::Add(OperatorA,OperatorB)
```

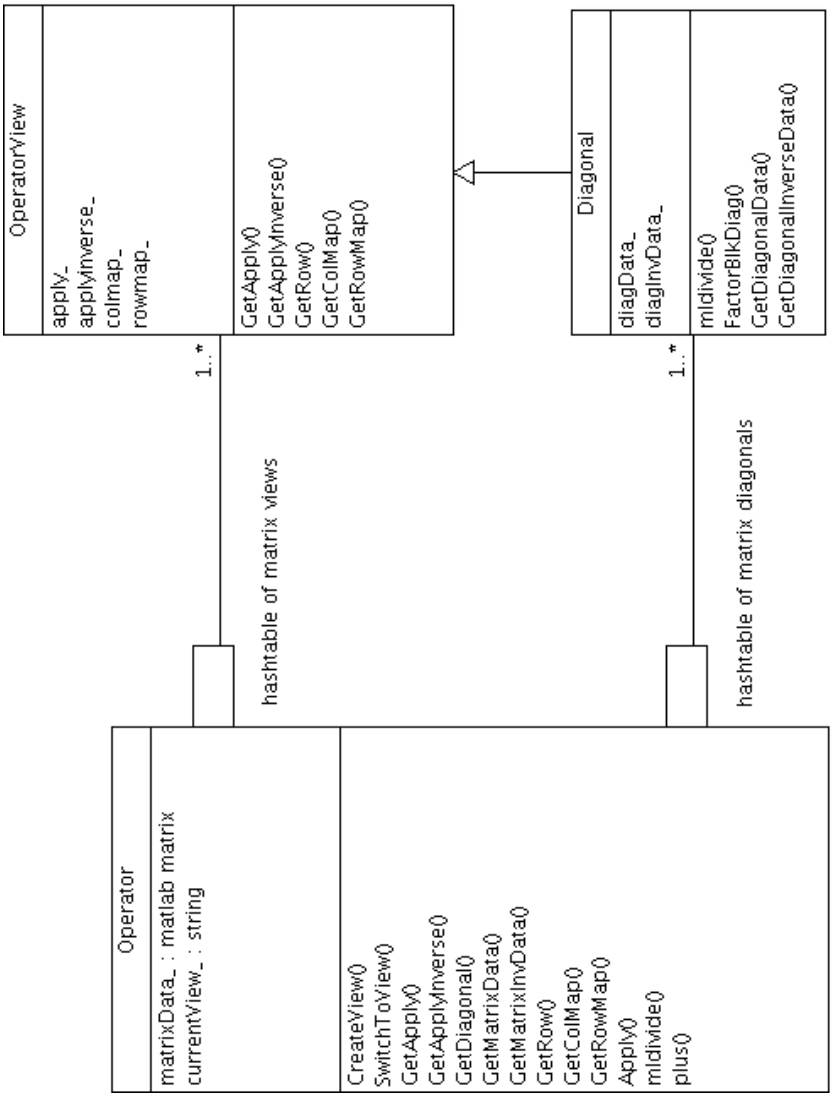

Figure 2: Class diagram of `Operator` classes.

```
  CommonView = View.GetCommonView(OperatorA.DefaultView(),
                                  OperatorB.DefaultView());
  OperatorA.AddView('tmpView',CommonView);
  OperatorB.AddView('tmpView',CommonView);

  oldAview = OperatorA.SwitchToView('tmpView');
  oldBview = OperatorB.SwitchToView('tmpView');

  \% loop to perform A+B
  for()

  end

  OperatorA.DeleteView('tmpView');
  OperatorB.DeleteView('tmpView');

  OperatorA.SwitchToView(oldAview);
  OperatorB.SwitchToView(oldBview);
end
```

For `ViewTable`, what we need is **just** an hash table to get and set `View` of an `Operator`.

2.2.3 The `Diagonal` class

In MÜLOO, we would like to be able to use several format of diagonal (block diagonal, point diagonal ...). So, the View mecanism is also implemented for `Diagonal`.

Remark 2. `ViewTable` *is currently a list of struct with three fields:*

- *viewLabel_*
- *matrixView_*
- *diagonalView_*

So, there is a direct relation between the view of a Diagonal and the view of the Matrix. I think that it is better if we manage two separate list: one for diagonal

view and one for matrice view. Very specific diagonal can be created for the smoothing process. In Smoother.m, a 'Collection' can be use to define how to group blocks together for the smoothing process.

2.3 Summary from the March 23rd, 2010 meeting

Key operations that could impacted by the `operator` format are

1. relaxation
2. matrix/matrix multiplication
3. matrix/matrix addition.
4. constraint manager

An `operator` in MÜLOO should satisfy the following:

1. It will have a point `getrow`, point diagonal, and associated data (e.g., eigenvalue estimate for $D^{-1}A$).
2. By default, it will be stored natively as a point matrix. If for some reason, we want block storage, it must be explicitly converted. Here is an example:
   (a) User supplies $Ax = b$, all stored in constant block form.
   (b) MÜLOO amalgamates A and aggregates on the point matrix (graph, really), yielding a tall skinny point matrix B.
   (c) B is factored via QR into two matrices, P^{tent} and N_H, both in point format.
   (d) If we want P^{tent} to be in block form, we must do an explicit conversion.
   (e) JJH: An RAP involves two point matrices and one block matrix. I guess the result should be a point matrix?

   One exception to this is that the result of multiplying or adding two block matrices should be a block matrix. This makes sense, as both operations occur in a linear algebra library outside of MÜLOO.
3. It may or may not have a block `getrow`, block diagonal, and associated data.

4. Determining whether an `operator` is compatible with another `operator` reduces to determining whether the row maps are compatible. Compatible means that the data distribution and ordering is the same for both maps. Compatibility checks should ideally be done in the linear algebra library (or in the linear algebra interface). JJH: I still do not think that *compatibility* is sufficient. Suppose I have two serial identity matrices, one stored as a point matrix, I_p, and one as a 2×2 block matrix, I_b. Further, suppose both have the natural ordering and contain the same number of point rows. If I want $I_p + I_b$, I must view I_b as a point matrix with a point row map, even though I_b's block row map is compatible with I_p's row map.

2.4 Summary from the August 24th, 2010 meeting, with the ZOLTAN team

The Zoltan team is also interested to write a thin layer between Epetra/Tpetra and their future package, ZULU[3]. We discussed whether it makes sense for us to collaborate on a general linear algebra interface. The needs of ZULU are focus on graph class and more limited than MÜLOO. It is not clear if writting in common the layer is of interest. So, above all, the plan is to summerize the capabilities we expect to get from such layer in writing (in this document) and pass the document to the Zoltan team.

Mike Heroux pointed out to Jonathan that an interface package already exist in Trilinos (Thyra). It does'nt fit your needs, but we should keep that in mind (for example, if we would like to release the code of this interface separatly). The Belos package also use a layer between the linear algebra package and itself. Needs of Belos are also very limited. It only needs a layer for vector objects and Operator object only have to provide an Apply method. The implementation of the layer uses Traits.

2.5 Summary from the August 24th, 2010 meeting

During this meeting, we discuss several issues with the design of the operator class on the basis of the document known as "10 remarks about the Operator class ...". Main issues concernes MÜLOO and how to interoperate with Epetra, Petra, IFPACK and TIFPACK. The discussion about these issues have been moved to

[3] Chris lobbies intensively the Zoltan team to name their next-gen package in line with MÜLOO. It's not clear if it worked.

the corresponding parts of this documents. For the record, here are the summary of minor problems we discussed during the meeting:

2.5.1 Old class diagram of `Operator` classes

2.5.2 MÜMAT GetRow/GetBlockRow accessors

```
nRowBlk = op.GetRowMap().NNodes();
nColBlk = op.GetRowMap().NNodes();

if op.GetRowMap().HasVariableBlkSize()

  VarRowBlkPtr = op.GetRowMap().Vptr();
  VarColBlkPtr = op.GetColMap().Vptr();
  for i=1:nRowBlk
    fRow = VarRowBlkPtr(i);
    lRow = VarRowBlkPtr(i+1)-1;

    for j=1:nColBlk
      fCol = VarColBlkPtr(j);
      lCol = VarColBlkPtr(j+1)-1;

      currentBlock = Amat(fRow:lRow, fCol:lCol);
    end % for j
  end % for i
end
```

Remark 3. *I think we should provide a GetRow/GetBlockRow accessors in* MÜMAT`Operator` *to rewrite such algorithms more like they will be written in* MÜLOO.

2.5.3 function handles for ApplyFunc/InvApply

Remark 4. *I don't understand why we use function handles for ApplyFunc/InvApply functions. In a object oriented design, we can instead implement an Operator subclass for each internal data storage.*

2.5.4 Views

Remark 5. *I propose to rename the class* `SingleViewOperator`

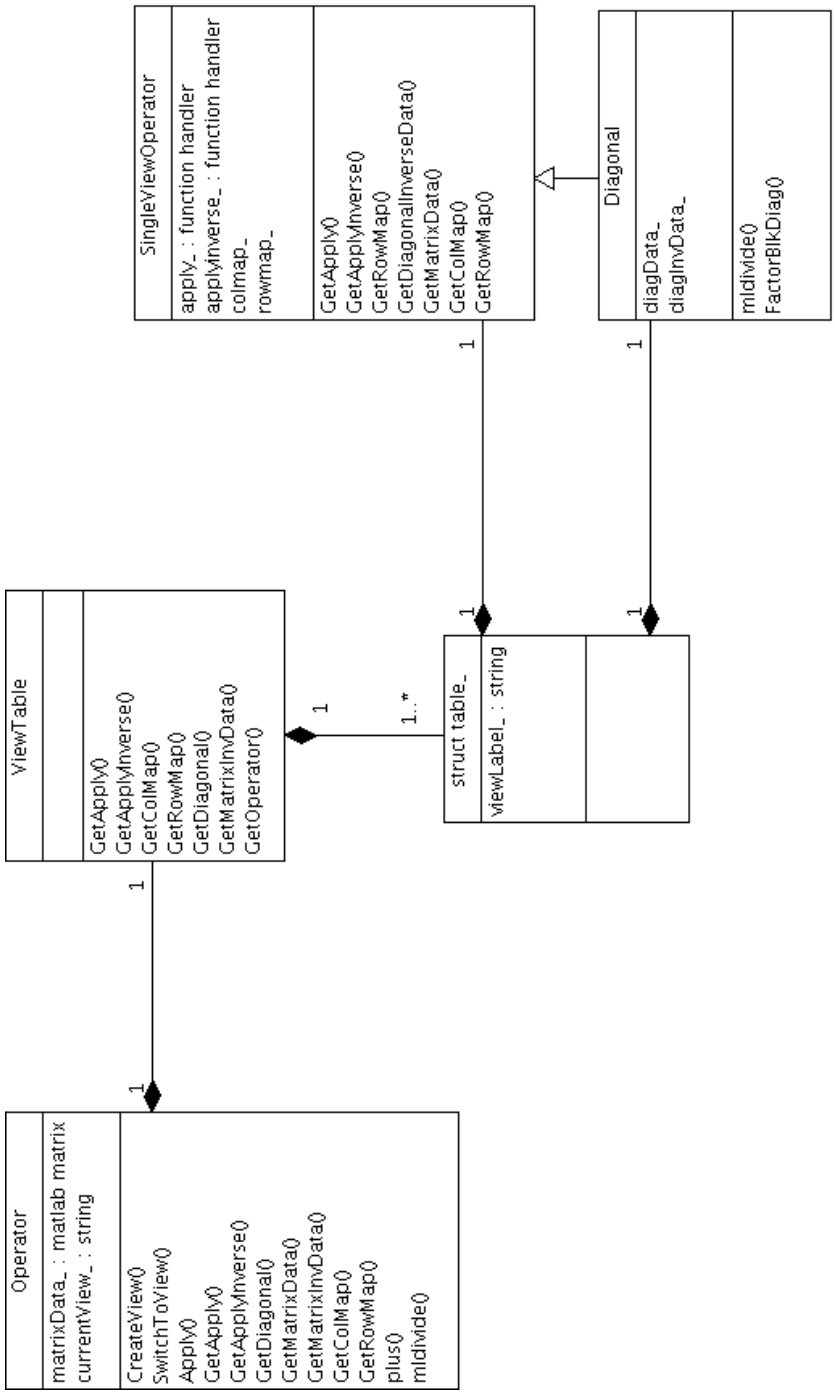

Figure 3: Class diagram of `Operator` classes.

⇒ ***has been renamed*** `OperatorView`**.**

Remark 6. `ViewTable` *should be only a very basic data structure. What we need is* ***just*** *an hash table to get and set* `View` *of an* `Operator`*. In my opinion,the ViewTable implementation is much too heavy: Each method of* `SingleViewOperator` *are reimplemented in* `ViewTable` *to provide a direct accessor to the* `SingleViewOperator` *object properties corresponding to the requested* `View`*.*

2.5.5 Diagonal

Remark 7. *I recently added the* `Diagonal` *class to* MÜMAT*. Initialy, we planned to use* `SingleViewOperator` *class to store diagonals but in my opinion, there is to much diagonal specific code in MueMat (the stuff previously in src/Diag for extracting, creating, inverting diagonals) and it makes more sense to have a* `Diagonal` *class.*

Remark 8. *In Smoother.m, a 'Collection' can be use to define how to group blocks together for the smoothing process. At the moment, this diagonal is stored as the current diagonal but a special view should be created for such specific diagonal (and there is not directely related matrix view for this diagonal).*

Remark 9. *Methods GetDiagnal() and ExtractDiag() was confusing so now, GetDiagonal() extracts the matrix if the diagonal doesn't exist. We can use the same mecanism for FactorBlkDiag(): this method can be called automatically when we use ApplyInverse(). One drawback of this mecanism is that size of objects grows silently (when we add views to operator, extract diagonal, store LU factors...) and it is not very good...*

3 MÜLOO

This section discuss the design issues which are specific to the C++ codes.

3.1 Design of the `operator` Class

3.1.1 The `Operator` class

An `Operator` is simply a matrix. In MÜMAT, the matrix data are natively stored as a MATLAB matrix (encapsulated in the `Operator` object) and you can access

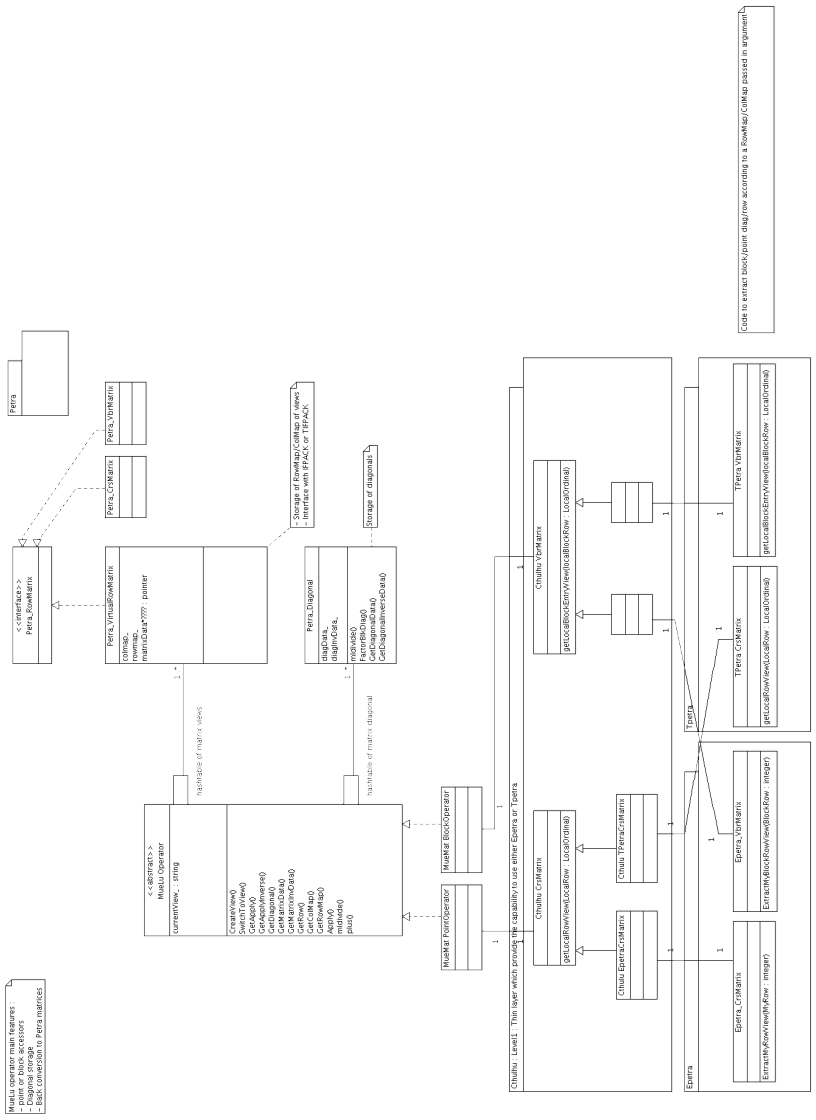

Figure 4: Class diagram of `Operator` classes.

directly to internal data storage by using the method GetMatrixData(). This accessor is used in MÜMAT for convenience and efficiency reasons.

In MÜLOO, matrices will be stored in point or block form (using Epetra or Tpetra CsrMatrix or VbrMatrix). Such format provides row accessors (ie: GetRow() or GetBlockRow()):

- GetRow(i) returns the list of non-zeros of the row *i* (as an array of columns indices and array of values).
- GetRow(i) returns the list of blocks of the row *i* (as an array of columns indices and an array of dense matrices).

In such format, matrix blocks are described by Map objects (`Epetra_Map` or `Tpetra_Map`).

Some MÜMAT algorithms mimic the fact that we only have such row accessors: These algorithms loops over the rows and blocks and extract blocks from the MATLAB matrix:

3.1.2 Additional remarks

Remark 10. *A thin layer to be able to use easily Epetra or Tpetra matrix is also of interest for other projects than* MÜLOO *and should be available as a standalone library. This implies to separate this functionality to other functionality of our future C++* `Operator` *class (and it implies also to do more work to support all methods of Petra matrices).*

Remark 11. *Same remark as remark 10 but for the multiview capability of our* `Operator` *class. Such functionality should be added directly to petra libraries (by specializing CSR and VBR matrix classes).*

3.1.3 The view mechanism

One goal of the `Operator` class is to be a thin interface layer between Epetra/Tpetra and MÜLOO (to allow us to be compatible with both libraries).

3.2 Interface between the linear algebra packages (Epetra/Tpetra) and MÜLOO

Why? - ¿ compatible with Epetra and Tpetra
Should the linear algebra interface consist of:

- A single base class operator with view capabilities. This base class will be used in MueMat. Can derive specializations for each storage format.
- ...

3.2.1 Matrix-Matrix multiply used in MueMat

$RAP = R(AP)$

1. point-point
2. point-block
3. block-block
   - with the same block structure
   - $(I - D^{-1}A)P$ where $D^{-}1$ is 3×3, A is 3×3 and P is 3×6
     $\Rightarrow (6 \times 3)$ multiplied by (3×3)

So, notion of compatible blocks... tbd
$D^{-1}A$

- point-point
- point-block
- block-point
- block-block

3.2.2 Requirement

Here are the summary of what we suppose to get from Epetra/Tpetra:

- Extraction of block diagonal and diagonal with overlap (Collection)
-

3.2.3 Important issues

Matrix-Matrix multiply in ?Petra.

3.3 Interface between IFPACK/TIFPACK and MÜLOO

A generic MueMat operator must be able to hand the underlying data to TIFPACK/IFPACK/AZTECOO

3.3.1 Important issues

- TIFPACK smoothers currently always extract point diagonals.

3.4 Summary of JG and JJH Discussion, 9/10/2010

As a reminder, the view mechanism is the mechanism which allows someone to access a MÜLOO matrix as if it is stored using another data storage format. For instance, a point matrix can be access as a block matrix (via a `GetBlockRow` method) or vice versa. In addition, a 2x2 block matrix can also be viewed as a 4x4 matrix in MÜLOO. In this discussion, we suppose that the view mechanism is **not** provide by TPETRA (or TPETRAEXT) and is instead implemented either in CTHULHU[4] or MÜLOO.

There are essentially two paths that we can take to interface MÜLOO and TIFPACK.

3.4.1 Path 1: TIFPACK and MÜLOO live in their own little worlds

In the first path, we suppose that TIFPACK takes as arguments either TPETRA point or block matrices, and provides (for both types of matrices) an implementation of point and block smoothers.

Here are some implications:

1. MÜLOO must provide TPETRA point/block matrices to TIFPACK (and other packages that use TPETRA). In most cases, we can just pass to TIFPACK the internal matrix we stored. But, if we want to be able to use 4x4 block smoothers to 2x2 matrices, we need to provide **fake** TPETRA **point/block matrices** to TIFPACK. Such fake matrices are just lightweight objects that

[4] CTHULHU is our future linear algebra layer between EPetra/TPetra and MÜLOO

respect the interface of EPETRA matrices and wrap method calls, i.e., they derive from the `Epetra_RowMatrix` interface.

2. If TIFPACK is passed a `Tpetra_CrsMatrix`, but the user wants a block smoother, then TIFPACK has to do its own internal GetRow conversion to create blocks and apply the block smoother. There is potential code duplication between MÜLOO and TIFPACK concerning the mechanism to create blocks.

3. There is only one implementation of block smoothers in IFPACK and this implementation is used for both point and block matrices. The block smoother is implemented using only the interface provided by `Epetra_RowMatrix`. Indeed, IFPACK uses the method `Epetra_RowMatrix.ExtractMyRowCopy()` to build its own version of blocks. The implementation of block smoothers for block matrices in IFPACK is very inefficient. If the matrix is stored natively as a block matrix, we should takes advantage of that in TIFPACK. In particular, a specicialized block smoother code must be written for block matrices.

4. Both TIFPACK and MÜLOO will need to extract (block) diagonals. Therefore, it make senses that TPETRA (or TPETRAEXT) provides a method to extract **block** diagonals. This will avoid code duplication between MÜLOO and TIFPACK. We should also avoid extracting the same diagonal more than once.

3.4.2 Path 2: TIFPACK (and others) uses CTHULHU

In this path, the view mechanism is implemented in CTHULHU. TIFPACK and MÜLOO use CTHULHU matrices. Here are some implications:

1. The interface between MÜLOO and TIFPACK are simplified. We don't need anymore to create fake TPETRA matrices. Other users of TIFPACK could provide TPETRA matrices, which would immediately be wrapped as CTHULHU matrices at no cost.

2. TIFPACK will use the view capabilities of CTHULHU. TIFPACK would not have to roll its own `GetRow` and `GetDiagonal` code.

3. TIFPACK could provide efficient block relaxation schemes, etc., if the matrix is natively stored as a block matrix. We don't need an implementation for point matrices because point matrices can be viewed as block matrices.

4. If TPETRA (or TPETRAEXT) doesn't provide a method to extract **block** diagonals, CTHULHU can provide it. In addition, we don't have to worry about multiple diagonal extraction because when the diagonal have been extracted, it is stored within the CTHULHU matrix.

Here are the main advantages if TIFPACK were to use CTHULHU:

1. TIFPACK can now be used with both EPETRA and TPETRA matrices.

2. TIFPACK would not have to roll its own code to manage point and block matrices effectively (i.e.: dynamic cast of RowMatrix to CRSMatrix or VBRMatrix to avoid row copies when row views are enough; only one implementation of each smoother etc.).

3. Any TRILINOS package, not just MÜLOO, can invoke TIFPACK with a blocking scheme other than the native one. No additional code in TIFPACK is needed for this feature.

Note that we still can provide a mechanism to convert CTHULHU to fake Epetra/Tpetra matrices, but this functionality is now optional.

3.4.3 Commonalities

1. MÜLOO will always use CTHULHU matrices internally.

2. CTHULHU will always contain adapters to Epetra/Tpetra point and block matrices.

3.4.4 Implication of Path 1/2 to the class diagram

In the first path, the CTHULHU layer can be dedicated to wrap Epetra/Tpetra matrices and to provide a common interfaces between these libraries and MÜLOO. In this case, the MÜLOO Operator handles switching views. It contains the hashtables and the current view. Each view will correspond to a particular row map (point or block), getrow, and diagonal. There are two implementations of the MÜLOO Operator interface: one for matrices stored internally as CRS matrices and one for block matrices. There is no reason to push the view mechanism down to CTHULHU if nobody uses it.

In the second path, CTHULHU handles switching of views. So, the hashtable (JG: or the whole MueMat operator ?) gets pushed down to the CTHULHU level.

For example, suppose we have a CTHULHU EpetraCrsMatrix. If in MÜLOO we want a block view, we invoke a method in CTHULHU, which remembers this.
JJH: This could have unintended side-effects if we're not careful. The CTHULHU matrix state will persist, even when the matrix is operated on by other Trilinos packages that use CTHULHU.

3.4.5 Discussion about advantage and drawback of each approach

- Ray is concerned about the fact that Path 2 can slow down the development of MÜLOO (for example, if TIFPACK developers are waiting code from CTHULHU). This is certainly a valuable argument to take into account. Interaction between MÜLOO and TIFPACK will be more complicated in Path 2.

- Path 2 have mostly advantages for TifPack developers and will be (ironically) more difficult to making accepted by TIFPACK developers (We force the use of CTHULHU!).

- The amount of work for us is similar in both paths. In a coding perspective, the main difference between Path 1 and Path 2 is where the view mechanism is located (in MÜLOO or CTHULHU). If we decide to put the view mechanism in CTHULHU, we have the choice of moving from Path 1 to Path 2 in the future.

- The mechanism to convert CTHULHU to fake Epetra/Tpetra matrices is not a critical feature. For TPETRA/TIFPACK, we can directly pass the underlying TPETRA matrices to TIFPACK (if TIFPACK smoothers are available for both TpetraCrsMatrix and TpetraVbrMatrix). The same is possible for the couple EPETRA/IFPACK. In fact, fake matrices are required for:

  - Applying block smoother with different blocking scheme than the native format of TpetraVbrMatrix (but only if TIFPACK don't have such functionality).
  - Using EPETRA matrices in TIFPACK or TPETRA matrices in IFPACK.

- If TIFPACK don't use CTHULHU and don't provide an efficient block relaxation scheme for block matrices, we can roll our own later (and use CTHULHU in TIFPACK ourselves if it simplified developement).

Questions:

- What is the current state of TIFPACK? Does TIFPACK developers plan to write an efficient block relaxation scheme for block matrices ?
- Do we need to plan a meeting with TIFPACK guys (at least to let them know about CTHULHU) ?
- Path 1 or Path 2 ? Or we postpone the decision for the moment ?

3.4.6 Chris' Comments

1. Why does the serial matrix-matrix multiply that Ray will work on need to use CTHULHU? Is this just to flesh out the CTHULHU API, or does the real multiply need to use CTHULHU?
2. There is a 3rd path. Namely, CTHULHU hides whether EPETRA or TPETRA is being used. Any block access, etc. is pushed down to the linear algebra level. JJH: My counter to this is that our pace of development is potentially constrained by what the TPETRA/EPETRA developers want/specify/object to.

3.4.7 Ray's Comments

Kernels for blocking

1. We want to buid a matrix graph associated with a block matrix even if the blocking does not match the original blocking in the matrix. This is useful for implementing Coalesce and it perhaps could be useful in a block matrix-matrix multiply. One good thing is that storage of the block matrix graph is often not too bad compared to the original matrix. For example, if we have a point matrix, the edge/vertex information would be relatively small for the associated block matrix graph corresponding to something like 4x4 blocks,
2. We want something like GetBlockRow(). This is relatively easy if we have a point matrix. It is somewhat more costly if we have a block matrix and we want to get a point row. This involves first figuring out which block row contains a point row. We could actually store something to make this fast. Then, there are a bunch of offsets to figure out for each nonzero block in this block row (and of course none of the data is contiguous).

3. A bit more tricky is that the matrix-matrix multiply might want to easily get a sub-block within a Block Row. This would implies that GetBlockRow() should really return a list of blocks. If the underlying format is a point matrix, this would mean copy data and reorganizing it into a list of blocks. One might be able to make this okay efficient if the block matrix graph is stored ... perhaps with a little auxilliary info.

4. This might mean that it would be useful to have cThUlHu commands to build auxilliary information to optionally make something more efficient and perhaps special commands to remove this auxilliary information if it is not needed.

At the lowest level, it looks like we want something like

1. BuildBlockGraph(Matrix,RowMap,ColMap) perhaps with some way to efficiently include dropping?

2. GetRow(Matrix,Row)

3. GetBlockRow(Matrix,BlkRow) which returns a list of blocks

4. MatVec

5. SubsetMatVec

6. OptimizeForGetBlockRow(Matrix)

7. RemoveOptimization(Matrix)

8. GetDiag(Matrix)

9. GetBlkDiag(Matrix)

I would guess that SubsetMatVec() in conjunction with GetBlkDiag() is a more efficient way to implement Gauss-Seidel than GetBlockRow().

We probably also need things like NNZ() and MaxNNZPerRow(). We might need these to work in a block sense?

Parallel?