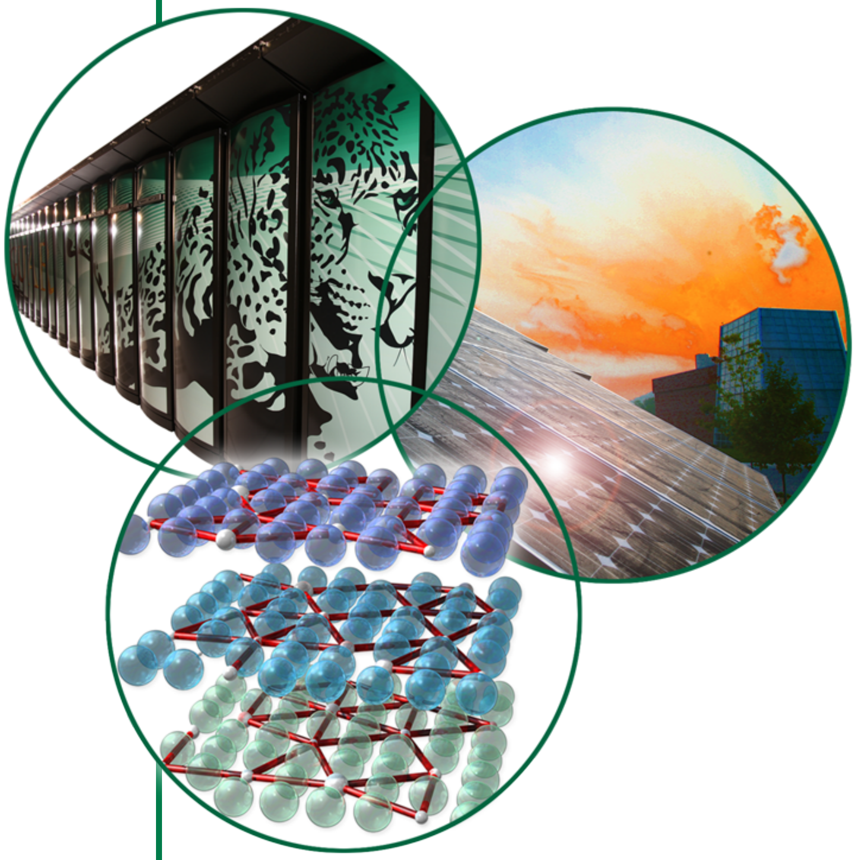
Incorporating Trilinos Data Classes into an Application

Chris Baker

Computational Engineering and Energy Studies

Oak Ridge National Laboratory, USA



Trilinos User Group 2011

Oct 31-Nov 3, 2011

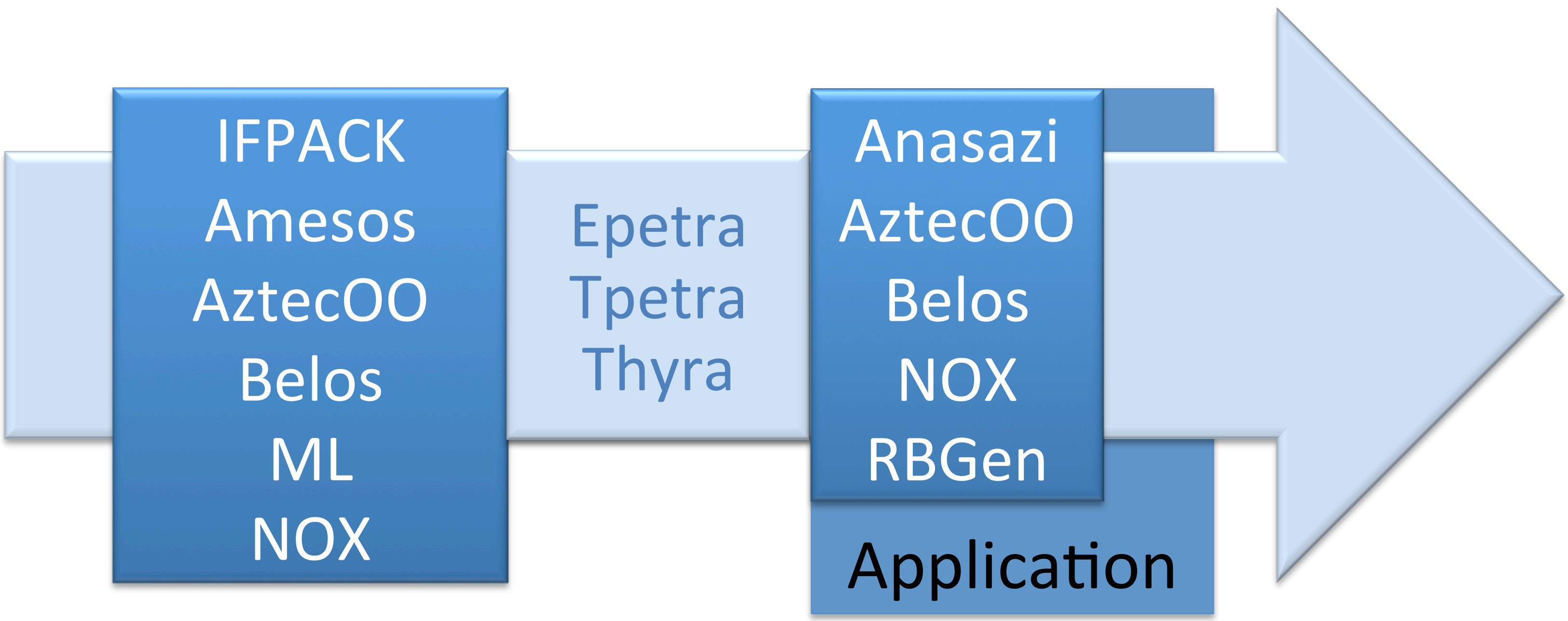
Sandia National Labs, Albuquerque, NM

U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Trilinos Overview

- **POV: Trilinos is all about operators.**
- **Operator abstraction is the heart of the solvers, and the main interface between applications and Trilinos.**



What is an Operator?

- **Operator** is a **Petra Object Model** concept
- **Operator** encapsulates the effect of some action (typically linear) from one vector into another.

Epetra

```
virtual Epetra_Operator::Apply(const Epetra_MultiVector &X,
                                     Epetra_MultiVector &Y) const;
```

Tpetra

```
virtual Tpetra::Operator<S>::apply(
        const Tpetra::MultiVector<S> &X,
              Tpetra::MultiVector<S> &Y,
        ...) const;
```

Thyra

```
virtual Thyra::LinearOpBase<S>::apply(...,
         const Thyra::MultiVectorBase<S> &X,
               Thyra::MultiVectorBase<S> *Y,
         ...) const;
```

Example Operators

- Direct operator implementations:
  - sparse matrix (Epetra/Tpetra)
  - dense matrix (Epetra/Tpetra)
  - direct solve (Amesos, IFPACK, Epetra/Tpetra)
- Indirect operator implementations:
  - iterator linear solve (AztecOO, Belos)
  - composition of operators (Epetra, Thyra)
- Application-specific:
  - ???

Operator Interface

- **Operator** defines three main methods:

```
void apply(const Vector x, Vector y);
Map  getDomainMap();
Map  getRangeMap();
```

- `apply()` obviously realizes the effect of the operator.
- The others describe the properties of the input/output vectors, namely, the domain and range of the operator.

```
x = createVector<S>( op->getDomainMap() );
y = createVector<S>( op->getRangeMap() );
op->apply( *x, *y );
```

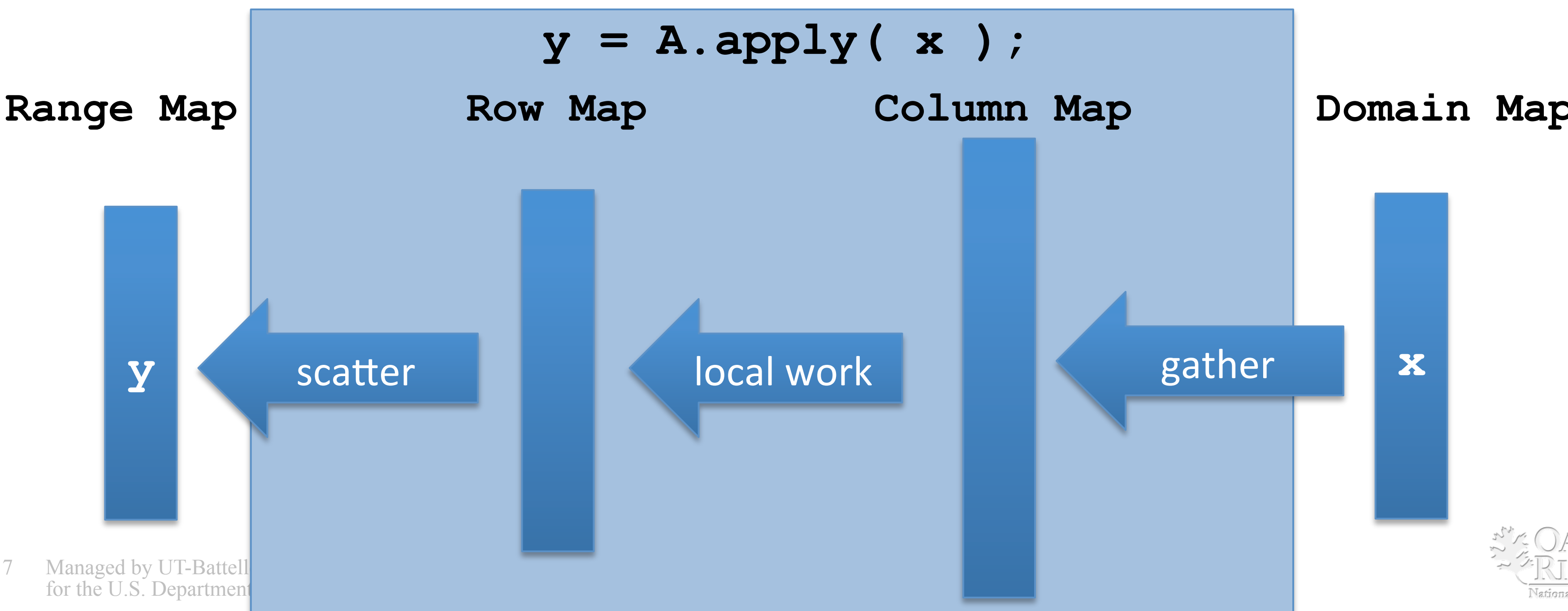
Whither the Map

- The **Petra Object Model** describes the construction and manipulation of distributed memory objects.
- The **Map** concept addresses two main concerns:
  - distribution of a global object across local memories
  - translation between global indices and local indices
- A related concept is the **Distributed Object**, an object that is distributed in this manner.
- A **Distributed Object** is distributed according to a **Row Map:**
  - examples: `Vector`, `MultiVector`, `CrsMatrix`, `CrsGraph`
- This describes a row distribution of the data.

Other Common Maps

In addition to Domain Map and Range Map, Operator objects often utilize two other Map objects:

- **Row Map** indicates which rows of the output are computed, a row distribution of the work.
- **Column Map** indicates the rows of the input vector necessary to perform that work, a function of the Operator and the Row Map



Supporting cast

- The gather/scatter in a communicating **Operator** is facilitated by **Import** and **Export** objects.
- These contain information needed to transfer data between **Distributed Objects** described by different **Maps**.
- These are constructed in advance, analyzing source and destination **Maps** for significant quantities:
  - remote indices, owners
  - local/permuted indices
  - total transfer size (for allocating buffers)

Impact on App Developers

- What does this mean for app developers?
- For current Epetra users moving to Tpetra, the concepts are the same.
  – The semantics and implementations are mostly the same, but translation of syntax is needed in many places.
- For developers new to Trilinos, you may need some understanding of these.
  – This is necessary, at the least, when instantiating vectors and defining operators.

Side by side: Creating Maps

- Initialize MPI, create a communicator
- Create a uniformly distributed map with contiguously allocated elements, query the list of elements

Epetra

```
MPI_Init(&argc,&argv);
Epetra_MpiComm comm( MPI_COMM_WORLD );
Epetra_Map map(NumGlobalElements, 0, comm);
const int NumMyElements = map.NumMyElements();
std::vector<int> MyGlobalElements(NumMyElements);
map.MyGlobalElements( &MyGlobalElements[0] );
```

Tpetra

```
Teuchos::GlobalMPISession mpiSession(&argc,&argv,...);
Platform &plat = Tpetra::DefaultPlatform::getDefaultPlatform();
RCP<const Teuchos::Comm<int> > comm = platform.getComm();
RCP<const Tpetra::Map<int> > map;
map = Tpetra::createUniformContigMap<int>(numGlobalElements, comm);
const size_t numMyElements = map->getNodeNumElements();
Teuchos::ArrayView<const int> myGlobalElements;
myGlobalElements = map->getNodeElementList();
```

Side by side: Creating Maps

- Create a map with specified number of elements
- Create a map with a specified list of elements

Epetra

```
Epetra_Map mapContig(numGlobal, numLocal, 0, comm);

std::vector<int> elemList(numLocal);
// ... fill elemList ...
Epetra_Map mapFromList(numGlobal, numLocal, &elemList[0], 0, comm);
```

Tpetra

```
RCP<const Tpetra::Map<int> > mapContig, mapFromList;
mapContig = Tpetra::createContigMap<int>(numGlobal, numLocal, comm);

Teuchos::Array<int> elemList(numLocal);
// ... fill elemList ...
mapFromList = Tpetra::createNonContigMap<int>(elemList(), comm);
```

Side by side: Vectors and MultiVectors

- Create **Vectors** or **MultiVectors** from maps
- Extract **Vectors** from **MultiVectors**
- Extract data pointers from **Vectors**

Epetra

```
Epetra_Vector vec(map);
Epetra_MultiVector mvec(map, numVecs);
for (i=0; i<numVecs; ++i) {
  Epetra_Vector * vecptr = mvec(i);
  double * vecdata       = mvec[i];
}
```

Tpetra

```
RCP< Tpetra::Vector<double> > vec;
vec = Tpetra::createVector<double>(map);
RCP< Tpetra::MultiVector<double> > mvec;
mvec = Tpetra::createMultiVector<double>(map, numVecs);
for (i=0; i<numVecs; ++i) {
  RCP< Tpetra::Vector<double> > veci = mvec->getVector(i);
  ArrayRCP<double> vecdata = veci->get1dViewNonConst();
}
```

Side by side: Writing an Operator

- A simple, non-communicating diagonal Operator

Epetra

```
class DiagOp : public Epetra_Operator {
  int Apply(const Epetra_MultiVector &X, Epetra_MultiVector &Y) {
    for (int v=0; v < X.NumVectors(); ++v)
      for (int i=0; i < X.MyLength(); ++i)
        Y[v][i] = diag_[i] * X[v][i];
    return 0;
  }
};
```

Tpetra

```
template <class T> class DiagOp : public Tpetra::Operator<T> {
  void apply(const Tpetra::MultiVector<T> &X, Tpetra::MultiVector<T> &Y,
             Teuchos::ETransp mode, Scalar alpha, Scalar beta) {
    ArrayRCP<ArrayRCP<const T> > xdat = X.get2dView();
    ArrayRCP<ArrayRCP<      T> > ydat = Y.get2dViewNonConst();
    for (int v=0; v < X.getNumVectors(); ++v)
      for (int i=0; i < X.getLocalLength(); ++i)
        ydat[v][i] = beta * ydat[v][i] + alpha * diag_[i] * xdat[v][i];
  }
};
```

Side by side: Using Anasazi/Belos

- **Switching Anasazi/Belos from Epetra to Tpetra requires only modifying the data initialization.**

Epetra

```
typedef double                 T;
typedef Epetra_MultiVector    MV;
typedef Epetra_Operator       OP;
RCP<OP> op = ...;
RCP<MV> init;
init = rcp( new Epetra_MultiVector(op->RangeMap(), blockSize) );
Anasazi::LOBPCGSolMgr<T,MV,OP> eigensolver( ... );
Belos::BlockCGSolMgr<T,MV,OP> linearsolver( ... );
```

Tpetra

```
typedef double                            T;
typedef Tpetra::MultiVector<double> MV;
typedef Tpetra::Operator<double>    OP;
RCP<OP> op = ...;
RCP<MV> init;
init = Tpetra::createMultiVector<double>(op->getRangeMap(), blockSize);
Anasazi::LOBPCGSolMgr<T,MV,OP> eigensolver( ... );
Belos::BlockCGSolMgr<T,MV,OP> linearsolver( ... );
```

Side by side: Communicating Operators

- **Communicating Operators with Import/Export**

Epetra

```
MyOp::MyOp(const Epetra_Map &rowMap,
           const Epetra_Map &rangeMap,
           const Epetra_Map &domainMap)
{
  rowMap_(rowMap);
  // ... build colMap_ from _rowMap and operator specifics ...
  Importer_     = new Epetra_Import(colMap_, domainMap);
  Exporter_     = new Epetra_Export(rowMap_, rangeMap);
  ImportVector_ = new Epetra_MultiVector(colMap_, blockSize);
  ExportVector_ = new Epetra_MultiVector(rowMap_, blockSize);
}

int MyOp::Apply(const Epetra_MultiVector &X, Epetra_MultiVector &Y)
{
  ImportVector_->Import(x, *Importer_, Insert);
  xp = (const double*) ImportVector_->Values();
  yp = (      double*) ExportVector_->Values();
  // ... apply operator from xp to yp[r] for each r in rowMap_ ...
  Y.Export(*ExportVector_, *Exporter_, Add));
  return 0;
}
```

Side by side: Communicating Operators

- Tpetra is conceptually the same... with caveats.

Tpetra

```
template <class T, class LO, class GO>
void MyOp<T,LO,GO,Kokkos::SerialNode>::apply(
          const Tpetra::MultiVector<T,LO,GO,Kokkos::SerialNode> &X,
                Tpetra::MultiVector<T,LO,GO,Kokkos::SerialNode> &Y,
          ...)
{
  importMV_->doImport(X, *importer, INSERT);
  ArrayRCP<ArrayRCP<const T> > xp = importMV_->get2dView();
  ArrayRCP<ArrayRCP<      T> > yp = exportMV_->get2dViewNonConst();
  // ... apply operator from xp to yp[r] for each r in rowMap_ ...
  Y.doExport(*exportMV_, *exporter, ADD);
}
```

- **POM** concepts direct the code to a similar outline.
- Support for multi-core/GPUs begins to highlight weaknesses in the grab-the-pointer-and-run model.

Addressing OpenMP Nodes: Epetra

- **New releases of Epetra support OpenMP for sparse matrix multiply and for vector operations; GPUs soon.**
- **Therefore, custom `Epetra_Operator` objects in an OpenMP-enabled build should also support OpenMP.**

```
int MyOp::Apply(const Epetra_MultiVector &X, Epetra_MultiVector &Y)
{
  ImportVector_->Import(x, *Importer_, Insert);
  xp = (const double*) ImportVector_->Values();
  yp = (      double*) ExportVector_->Values();
#ifdef EPETRA_HAVE_OMP
#pragma omp parallel for ...
  // ... apply operator from xp to yp[r] for each r in rowMap_ ...
#else
  // ... apply operator from xp to yp[r] for each r in rowMap_ ...
#endif
  Y.Export(*ExportVector_, *Exporter_, Add));
  return 0;
}
```

Addressing Generic Nodes: Tpetra

- **Previous slide showed all template parameters in `Tpetra::Operator`**

`Tpetra::Operator<T, LO, GO, Node>`			
`T`	**`LO`**	**`GO`**	**`Node`**
Scalar field over which the operator applies	Ordinal type for local indices	Ordinal type for global indices	Kokkos node for shared-memory parallel node

- **Scalar type is generic, supporting broad capability.**
- **Ordinals templated separately to simultaneously maximize efficiency and capability.**
- **Node type is a template parameter in order to utilize Kokkos template meta-programming shared-memory API.**

Impact of Templated Classes on Apps

- Templated classes force apps into an immediate decision:
  - whether or not to use a generic programming approach?
- Option #1: utilize typedefs to hide templates
  - app avoids templates, but is hard-coded to this choice

```
typedef Tpetra::Map<long,int>                  Map;
typedef Tpetra::Vector<double,long,int>        Vec;
typedef Tpetra::CrsMatrix<double,long,int>     Mat;
typedef Tpetra::Operator<double,long,int>       Op;
```

- Option #2: embrace templates, suffer the consequences

```
template <class Scalar>
void everyMethodInMyWholeApplication(Operator<Scalar> &tedium) {
  // ... insert syntactic sugar here ...
}
```

Tpetra::HybridPlatform

- Encapsulate "main" in a templated class method:

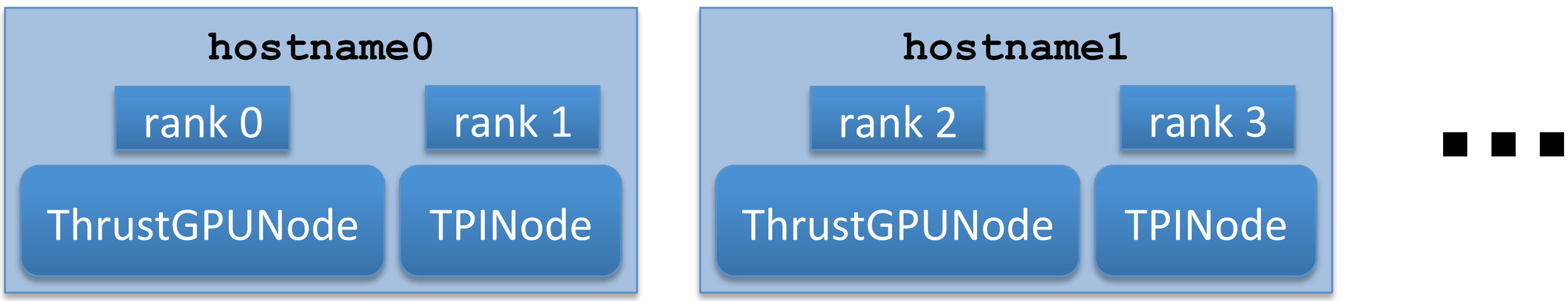
```
template <class Node> class myMainRoutine {
  static void run(ParameterList &runParams,
                  const RCP<const Comm<int> > &comm,
                  const RCP<Node> &node) {
    // ... do something interesting ...
  }
};
```

- `HybridPlatform` maps the communicator rank to the Node type, instantiates a node and the user routine.
  - `main()` becomes boilerplate code.

```
int main(...) {
   Comm<int>      comm          = ...
   ParameterList machine_file = ...
   // instantiate appropriate node and myMainRoutine
   Tpetra::HybridPlatform platform( comm , machine_file );
   platform.runUserCode< myMainRoutine >();
   return 0;
}
```

HybridPlatform Machine File

round-robin assignment	interval assignment	explicit assignment	default
%M=N	[M,N]	=N	default




```
<ParameterList>
  <ParameterList name="%2=0">
    <Parameter name="NodeType"       type="string" value="Kokkos::ThrustGPUNode"/>
    <Parameter name="Verbose"        type="int"    value="1"/>
    <Parameter name="Device Number"  type="int"    value="0"/>
    <Parameter name="Node Weight"    type="int"    value="4"/>
  </ParameterList>
  <ParameterList name="%2=1">
    <Parameter name="NodeType"       type="string" value="Kokkos::TPINode"/>
    <Parameter name="Verbose"        type="int"    value="1"/>
    <Parameter name="Num Threads"    type="int"    value="15"/>
    <Parameter name="Node Weight"    type="int"    value="15"/>
  </ParameterList>
</ParameterList>
```

What about Parallel Tpetra Operators?

- These can be written in the same way as Tpetra does:
  - Kokkos Shared-Memory Parallel Node API

```
template <class T, class LO, class GO, class Node>
void MyOp<T,LO,GO,Node>::apply(
          const Tpetra::MultiVector<T,LO,GO,Kokkos::SerialNode> &X,
                Tpetra::MultiVector<T,LO,GO,Kokkos::SerialNode> &Y,
          ...) {
  RCP<Node> node = X.getRowMap()->getNode();
  importMV_->doImport(X, *importer, INSERT);
  for (int v=0; v < X.getNumVectors(); ++v) {
    ArrayRCP<const T> xp = importMV_->getLocalMV()->getValues(v);
    ArrayRCP<T> yp;
    yp = exportMV_->getLocalMVNonConst()->getValuesNonConst(v);
    MyKernel<T,LO> kern(xp,yp,...);
    node->parallel_for(0, X.getLocalLength(), kern);
  }
  Y.doExport(*exportMV_, *exporter, ADD);
}
```

- All but the kernel-specific part is boilerplate code...

Kokkos Parallel Constructs

- Parallel for: execute loop iterations in parallel
- User-defined struct (*work-data pair*) contains:
  - the necessary data and `execute(int iter)`
- Parallel reduce: reduce implicit set of elements in parallel via user-specified associative binary operation
  - `typedef ReductionType`
  - `ReductionType identity()`
  - `ReductionType generate(int i)`
  - `ReductionType reduce(ReductionType a, ReductionType b)`
- Template meta-programming fuses generic loop skeleton with user data and kernel specifications.

```
Node::parallel_for   <WDP>(int beg, int end, WDP args);
Node::parallel_reduce<WDP>(int beg, int end, WDP args);
```

Kokkos parallel_for example

- Consider simple vector axpy: $y = \alpha * x + y$

```
template <class Scalar>
struct AxpyOp {
  Scalar alpha;
  const Scalar *x;
  Scalar *y;
  inline void execute(int i) {
    y[i] += alpha * x[i];
  }
};
```

```
AxpyOp<double> daxpy( ... );
Node::parallel_for(0,N,daxpy);

AxpyOp<complex<float> > caxpy( ... );
Node::parallel_for(0,N,caxpy);
```

Kokkos parallel_reduce example

- **Consider real-valued vector inner product:** $\alpha = x^T y$

```
template <class Scalar>
struct DotOp {
  const Scalar *x, *y;
  typedef Scalar ReductionType;
  Scalar identity() { return 0; }
  Scalar generate(int i)   {
    return x[i]*y[i];
  }
  Scalar reduce(Scalar a, Scalar b) {
    return a+b;
  }
};
```

```
DotOp<float> fdot( ... );
float f   = Node::parallel_reduce(0,N,fdot);

DotOp<qd_real> qddot( ... );
qd_real q = Node::parallel_reduce(0,N,qddot);
```

Tpetra RTI Operator Methods

- Tpetra Reduction/Transformation Interface provides convenience methods/macros for applying user Kokkos kernels to Tpetra Vectors/MultiVectors.

```
RCP< Tpetra::Map<LO,GO,Node> > domMap, rngMap, rowMap, colMap;
RCP< Tpetra::Import<LO,GO,Node> > importer = ...;
RCP< Tpetra::Export<LO,GO,Node> > exporter = ...;
MyKernel<T,LO> kern(...);
RCP< Tpetra::Operator<T,LO,GO,Node> > op;
op = Tpetra::RTI::kernelOp<T>(kern,domMap,rngMap,importer,exporter);
op->apply(x, y);
```

- Also wrappers for applying general functors.
  - e.g.: simple diagonal operator using a C++11 lambda function

```
RCP< Tpetra::Map<LO,GO,Node> > map;
RCP< Tpetra::Operator<T,LO,GO,Node> > op;
op = Tpetra::RTI::binaryOp<T>( [](T, T x) {return 2.0 * x;} , map );
op->apply(x, y);
```

Tpetra RTI Vector Methods

- Set of stand-alone non-member methods, e.g.:
  - `unary_transform<UOP>(Vector &v, UOP op)`
  - `binary_transform<BOP>(Vector &v1, const Vector &v2, BOP op)`
  - `reduce<G>(const Vector &v1, const Vector &v2, G op_glob)`
- This level provides maximal expressiveness, but coarser levels brings convenience.

```
// single-prec dot() with double-prec accumulator using custom kernels
result = Tpetra::RTI::reduce( *x, *y,  myDotProductKernel<float,double>() );
// Or a composite adaptor and standard functors
result = Tpetra::RTI::reduce( *x, *y,
                              reductionGlob<ZeroOp<double>>(
                                    std::multiplies<float>(),
                                    std::plus<double>()) );
// Or using inline functors via C++11 lambda functions
result = Tpetra::RTI::reduce( *x, *y,
                               reductionGlob<ZeroOp<double>>(
                               [](float x, float y)  {return x*y;} ,
                               [](double a, double b){return a+b;} );
// Or using a convenience macro to generate all of that
result = TPETRA_REDUCE2( x, y,   x*y, ZeroOp<float>, std::plus<double>() );
```

Easy parallel algorithm development

- Inline templated hybrid-parallel conjugate gradient.
  - Fun game: Find the MPI or threading!

```
for (k=0; k<numIters; ++k) {
  A->apply(*p,*Ap);                                // Ap = A*p
  S pAp = TPETRA_REDUCE2(
              p, Ap,
              p*Ap, ZeroOp<S>, plus<S>() );        // p'*Ap
  const S alpha = rr / pAp;                        // alpha = r′*r/p′*Ap
  TPETRA_BINARY_TRANSFORM(
              x, p,
              x + alpha*p );                       // x = x + alpha*p
  S rrold = rr;
  rr = TPETRA_BINARY_PRETRANSFORM_REDUCE(
              r, Ap,                               // fused kernels
              r - alpha*Ap,                        //   r - alpha*Ap
              r*r, ZeroOp<S>, plus<S>() );         //   sum r'*r
  const S beta = rr / rrold;                       // beta = r′*r/old(r′*r)
  TPETRA_BINARY_TRANSFORM(
              p, r,
              r + beta*p);                         // p = z + beta*p
}
```

Conclusion

- Tpetra and Epetra use cases are very similar.
- Capabilities have diverged somewhat, although many of the same issues exist regarding programming model.
- Understanding these classes is critical to leveraging most Trilinos solver libraries.
- Incorporation of these classes has application value independent of downstream package use.
- Still experimenting with programming models for multi-core platforms.