Scientific Programming 20 (2012) 115–128
DOI 10.3233/SPR-2012-0349
IOS Press

Tpetra, and the use of generic programming in scientific computing

C.G. Baker [a,*] and M.A. Heroux [b]

[a] *Computational Engineering and Energy Science Group, Oak Ridge National Laboratory, Oak Ridge, TN, USA*
E-mail: bakercg@ornl.gov
[b] *Department of Scalable Algorithms, Sandia National Laboratories, Albuquerque, NM, USA*
E-mail: maherou@sandia.gov

Abstract. We present Tpetra, a Trilinos package for parallel linear algebra primitives implementing the Petra object model. We describe Tpetra's design, based on generic programming via C++ templated types and template metaprogramming. We discuss some benefits of this approach in the context of scientific computing, with illustrations consisting of code and notable empirical results.

Keywords: Generic programming, scientific computing, template metaprogramming, shared-memory programming, distributed-memory programming, many-core computing

1. Introduction

Tpetra [33] is a package in the Trilinos project [32] implementing the Petra Object Model (POM) [14,17]. The POM is an abstract model describing the construction and interaction of parallel, distributed-memory matrices and vectors. This object model currently has three implementations: Epetra [33], a double-precision implementation emphasizing stability, scalability, portability, and interoperability with legacy C/FORTRAN codes; Jpetra [33], a Java implementation; and Tpetra, an implementation focusing on generic programming and multi-core/many-core node support.

Tpetra, in its current release as part of Trilinos 10.8, supports the following features:

- parameterization by type of scalar and ordinal fields, allowing varying precision, capability, storage and performance, as needed by the algorithm, application or platform;
- suitability for mixed-precision algorithms, with support for mixed-precision primitives and structures;
- usage of the Trilinos/Kokkos [7,8] shared-memory node API, with support for any parallel node supported by the API, including serial, CUDA [23] (via Thrust [19]), Intel TBB [20,26], Pthreads [31] (via the Trilinos ThreadPool package) and OpenMP [24];[1] and
- exposure of the underlying parallel node API, supporting hybrid (distributed + shared) parallel execution of user-provided kernels.

This paper discusses the Tpetra design, detailing these features and examining the benefits and trade-offs of a generic programming paradigm in the context of scientific computing. The audience of the paper is scientific computing practitioners: scientific library developers, domain specialists, numerical analysts and application programmers. We assume familiarity with C++ and its syntax, in general, and object-oriented and generic programming paradigms, in particular. Appropriate references for these concepts include [29,34,35] and many other texts.

Section 2 discusses the Tpetra design in the context of the Petra object model and other Trilinos packages. Section 3 details the use and benefits of the generic programming approaches employed in Tpetra. Examples are interspersed throughout, to illustrate the specifics of the discussion and provide empirical motivation.

[*] Corresponding author: Christopher G. Baker, Computational Engineering and Energy Science Group, P.O. Box 2008 MS6003, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6003, USA. E-mail: bakercg@ornl.gov.

[1] OpenMP support is finished in the developer branch, slated for released with 11.0; a preliminary version is available from the developers.

2. Design of Tpetra

The Tpetra package was designed to implement the Petra Object Model [14,17]. This programming model was first implemented in the Epetra package, which serves as the foundation for a majority of algorithm development in Trilinos. As such, implementing the POM leverages a proven model and eases the code migration for developers already familiar with Epetra. Having said that, the generic programming paradigm requires some considerations, and Tpetra's goal of supporting a hybrid-parallel programming environment (shared-memory nodes communicating via a distributed/message-passing layer such as MPI) means that Tpetra is not simply a "templatization" of Epetra. Additionally, at the time of Epetra's implementation and release (circa 2003), compiler support for C++ templates was not robust enough to allow sophisticated use of this language feature, especially on many scientific platforms. Therefore, Tpetra's current implementation (begun in 2008) is able to leverage nearly a decade of improved language support, as well as advances in the state of software engineering.

2.1. *Teuchos memory management*

One of Tpetra's most notable departures from Epetra is the use of automatic memory management classes from the Teuchos package [33]. These classes allow developers to replace explicit calls to **new** and **delete** with higher-level templated C++ classes that encapsulate and protect the allocations. In addition, raw pointer and array accesses are replaced by run-time checked operations (though many checks are disabled in a release/optimized build). For example, an array of floating point number that would be represented in primitive C/C++ as follows:

```
void someMethod(double *A, int ALEN) {...}
```

may instead be treated like so:

```
void someMethod(Teuchos::ArrayView<double> A) {...}
```

where the pointer and length are encapsulated in the ArrayView object. Traversal is done via the iterator associated with this class:

```
for ( iter = A.begin(); iter != A.end(); ++iter)
    *iter = someValue;
```

Table 1

Teuchos memory management classes

Class	Semantics	Description
Array<T>	std :: vector<T>	Dynamically-allocated, dynamic length container, with entries of type T
Ptr<T>	T*	Safe pointer
RCP<T>	boost :: shared_ptr	Safe and reference-counted pointer
ArrayView<T>	T[] and length	Safe view of an array
ArrayRCP<T>	n/a	Safe and reference-counted array

where each access of the iterator under a debug build is checked for bounds errors and dangling references. However, under a release build, the code is compiled down to raw pointers, so that the overhead is minimal. The significant Teuchos memory management classes are listed in Table 1, in order to advertise the capability of this package and to provide explanation for their occurrence in any code examples in this paper.

A detailed description of the Teuchos memory management classes, their implementation and their use cases is not in the scope of this paper; see the reference [13] for more information. However, it is worth pointing out that this is a significant departure from many scientific libraries. While the memory management classes have been designed to avoid any unnecessary runtime overhead in optimized builds, there is a significant amount of developer overhead associated with replacing all raw pointers with these classes. The benefit, however, is that Tpetra classes contain robust controls against common memory management errors. As Tpetra classes sit at the bottom of what is often a deep and convoluted software stack, it is critical that these errors are avoided, both in user application of Tpetra classes and its internal development as well.

2.2. *Kokkos node API*

The Petra Object Model, and the initial release of Epetra, were concerned with high-performance computing on distributed-memory machines. As such, there was initially no concern or support for shared-memory programming, though the most recent release of Epetra supports OpenMP on shared memory nodes [16]. As a new development effort with no requirements on backwards compatibility, Tpetra was designed to support a hybrid parallel programming model, where a distributed-memory, message-passing interface is used to communicate among a set of shared-memory parallel nodes. In order to maximize the utility from developing this shared memory capability by making it available to other Trilinos packages and software efforts, it was placed into a separate Trilinos package called Kokkos and is referred to as the Kokkos Node API [7,8].

This API was primarily designed to handle two challenges associated with portable numerical linear algebra kernels on current multi-core/many-core processors:

(1) Portable, parallel execution of user kernels. In particular, we are interested in the parallel for loops and parallel reductions that comprise the bulk of linear algebra operations.
(2) Recognition and management of distinct memory spaces associated with accelerators, e.g., general purpose graphics processors (GPUs), Cell multiprocessors, and Intel MIC ("Knights Corner").

As such, the Kokkos Node API comprises two main components. The first is the Kokkos memory model, describing methods for allocating memory regions suitable for shared-memory parallel computing (so-called *parallel compute buffers*) and transferring data between compute buffers and host memory.

The second component of the API is the parallel compute model. This is the interface by which user kernels are executed, in parallel, on the underlying shared-memory architecture. The goal of this effort was to enable write-once, run-anywhere kernels. The need to perform this inside the structures of C++ and while supporting templates favored the use of template metaprogramming. The mechanism works as follows:

(1) An API user writes a stateless, serial kernel; for example, the body of an AXPY (alpha-x-plus-y) operation: $y = \alpha * x + y$, for vectors x and y and a scalar α:

```
struct DAXPY {
  double alpha;
  const double *x;
  double *y;
  inline void execute(int i) { y[i] += alpha*x[i]; }
};
```

(2) An API developer writes a skeleton for a parallel for loop for a particular shared memory architecture; for example, Kokkos::SerialNode:

```
class SerialNode {
  template <class Kernel>
  void parallel_for (int beg, int end, Kernel k) {
    for (int i=beg; i < end; ++i) k.execute(i);
  }
};
```

(3) At compile time, the compiler fuses the kernel and the node, producing a shared-memory parallel DAXPY on the given architecture:

```
void SerialNode<DAXPY>::
parallel_for (int beg, int end, DAXPY op) {
  for (int i=beg; i < end; ++i)
    op.y[i] += op.alpha * op.x[i];
}
```

(4) At run time, user code will ask the node to allocate and initialize a parallel compute buffer, and then execute the AXPY kernel in parallel using the node object:

```
Kokkos::OpenMPNode node;
DAXPY op;
op.x = node.allocBuffer<double>(10);
op.y = node.allocBuffer<double>(10);
op.alpha = 2.0;
  // perform y = y + 2.0 * x
node.parallel_for (0, 10, op);
```

The API defines the syntax and semantics for both the nodes and the user kernels. This arrangement is preferable as it performs a separation of concern, between the library/application developer familiar with the kernels and the Node API developer familiar with a particular shared-memory node (which may utilize its own development API). Currently, two parallel skeletons are provided by the API: parallel for and parallel reduce.

One consequence of this is that the template metaprogramming approach is a compile-time polymorphism approach, which requires that the node type must be known at compile time. For this reason, the Tpetra design is such that all classes in Tpetra are templated on a Kokkos node, which is responsible for allocating compute buffers and executing the kernels needed by the Tpetra objects.

More information is available from the Kokkos website [33] or one of the following presentations [3,4,8].

2.3. *Petra Object Model*

The Petra Object Model, as initially defined in the implementation of Epetra, is concerned with the definition of distributed-memory SPMD (single-program/multiple-data) parallel linear algebra primitives. The POM provides a set of utility classes and methods, a set of linear algebra objects, and methods to interact with these linear algebra objects for the purpose of implementing numerical algorithms on a distributed-memory parallel system. The POM defines the following abstractions:

- Map defines a list of global elements, the distribution of global elements across a system, and the mapping between global elements and local elements on a particular node of the system.
- Distributed Object is an object, described by a Map, that is distributed and, potentially, redistributable.
- Import/Export are data structures containing the information necessary to move data between to Distributed Objects, defined by a source Map and a destination map.
- Vector is a distributed vector, representing an element of a vector space in the linear algebraic sense.
- MultiVector is a collection of vectors. This is a first-class object in the POM (and most of Trilinos), intended to provide optimal performance for block algorithms.
- Operator is an abstraction for a linear operator, encapsulating the application of the operator from one Vector/MultiVector into another.

In the Epetra implementation of the POM, the global and local indices are stored as **int** values (typically a 32-bit signed integer), and the scalars associated with Vector and MultiVector objects are stored as type **double** (typically a 64-bit floating point number). In Tpetra, these same abstractions are realized, but the underlying data components are templated. The list of template common template parameters in Tpetra is contained in Table 2. Some of the more commonly used Tpetra classes are listed in Table 3.

Table 2
Tpetra template class parameters

Parameter	Abbreviation	Description
LocalOrdinal	LO	Local element indices/labels
GlobalOrdinal	GO	Global element indices/labels
Scalar	S	Scalar field for arithmetic values, e.g., of vectors and matrices
Node	N	Kokkos node supporting a class

Table 3
Tpetra template classes

Map<LO,GO,N>	Map from global to local ordinals, dictating partitioning
Import<LO,GO,N>	Database for importing data based on requested elements
Export<LO,GO,N>	Database for exporting data based on available elements
DistObject<Packet, LO,GO,N>	Abstract base class for distributed/redistributable objects of type Packet, described by a Map<LO,GO,Node>
Vector<S,LO,GO,N>	Vector over field of type S
MultiVector<S,LO,GO,N>	MultiVector of over field of type S
Operator<S,LO,GO,N>	Abstract base class for linear operators over Vector<S,LO,GO,N>
CrsMatrix<S,LO,GO,N>	Compressed-sparse-row sparse matrix class with indices of type LO and values of type S
CrsGraph<LO,GO,N>	Compressed-sparse-row graph class with indices of type LO

3. Generic programming in Tpetra

The Tpetra library utilizes generic programming techniques to multiple ends. For the purpose of discussion, a distinction can be made, for example, between templated data (e.g., the scalars contained in a vector), templated operations (e.g., the arithmetic type associated with a mathematical operation), and the use of template metaprogramming (e.g., arithmetic operations associated with a comparison or reduction). In this section, however, with the goal of best motivating Tpetra's design to an audience of scientific programmers, we will distinguish the generic programming features in Tpetra according to their use case in scientific programming.

3.1. *Templated data for capability and performance*

One of the most common use case of templates in C++ involves template parameters dictating the data members of a so-called *template class*. The pedagogical examples are the template containers in the C++ Standard Template Library (STL). For example, std :: vector<**double**> and std :: list <**int**> respectively refer to a

STL vector of double-precision floating point numbers and a STL list of integers.

The Tpetra data classes outlined in Section 2 are each defined according to multiple template parameters. For example, the class Tpetra :: Map implements a POM Map abstraction and is parameterized on three types: a local ordinal type, a global ordinal type, and a Kokkos node. Table 2 lists the common template parameters utilized in Tpetra, and Table 3 lists the Tpetra template classes parameterized in this way. The design decision in Tpetra is to represent all resource intensive data as template types, the goal being extended capability of the code, improved portability and increased performance.

By templating the ordinal types used to label elements and indices, the capability of the Tpetra library is extended, with respect to the standard types – 32-bit integers and 64-bit floating point numbers. By distinguishing between the data type of local and global coordinate indices, the user is allowed to grow the GlobalOrdinal – and therefore, the global problem size – without necessarily growing the storage associated with local indices. This should be contrasted against a single ordinal type, perhaps decided before building the library, but fixed at that point. In the case of a sparse matrix, the smaller LocalOrdinal type reduces the footprint of the associated sparse graph, and therefore the bandwidth necessary to process the entire matrix; as sparse matrix–vector multiplication is typically a bandwidth-constrained operation, the result is that a smaller LocalOrdinal will often result in faster matrix–vector multiplications. Of course, the reduced footprint is inherently valuable, such as in situations where the matrix data is close to exceeding the system memory. Similarly, "smaller" scalar types results in a more efficient utilization of memory resources, as well as arithmetic resources. One can imagine a scenario where a sparse matrix with double the number of total nonzeros (corresponding, perhaps, to a physical discretization of a finer resolution) can be applied in the same amount of time, by reducing by half the associated data.

Of course, selection of the scalar and ordinal type is useful for extending the capability of the software as well. Larger local and global ordinal types allow for larger problems to be solved; the standard choice of 32-bit integer currently limits global problem sizes to between 2 and 4 billion entries (depending on signed/unsigned encoding) for many existing scientific libraries. On even modest clusters, this limitation in the software may take effect long before the resources of the computer are exceeded. Furthermore, flexibility in determining the scalar field of matrices and vectors (and, therefore, the arithmetic associated with the algorithms they realize) can be very useful. Examples of useful scalar types include:

- complex-valued types such as std :: complex<T>, enabling the direct implementation of complex-valued algorithms;
- automatic differentiation types [25,33], enabling automatic computation of derivatives and sensitivities;
- extended precision types, e.g., QD's **qd_real** and **dd_real** [18] and ARPREC's **mp_real** [1,2]; and
- utility types, e.g., flop-counting and shadow types (dual-scalar types used for evaluating the effects of different precision levels).

As an example, Table 4 gives the timings and solution accuracies for extended precision linear solves using the Tpetra templated linear algebra primitives and the Belos templated Block Conjugate Gradient iterative linear solver [33]. The source code for this example is in the Trilinos repository in the file:

```
packages/belos/tpetra/test/BlockCG/
test_bl_cg_hb_multiprec.cpp
```

It should be noted, this is possible because the Belos linear solvers, as well as the Anasazi eigensolvers [9, 33], are templated on the scalar type and the underlying linear algebra objects. Performing a linear solve in quad-double extended precision is as simple as instantiating a solver

```
Belos :: PseudoBlockCGSolMgr<
   qd_real,
   Tpetra :: MultiVector<qd_real,int>,
   Tpetra :: Operator<qd_real,int>
> solver ;
```

and calling the solve routine. This allows these solvers to be portable across linear algebra component libraries and scalar/ordinal types.

Table 4
Belos linear solve using QD for BCSSTK18 [15], $N = 11\text{k}$

Scalar	**float**	**double**	**dd_real**	**qd_real**
Solve time (s)	2.6	5.3	29.9	76.5
Solution accuracy	10^{-6}	10^{-12}	10^{-24}	10^{-48}

3.2. Reduction/transformation interface

Tpetra currently provides support for shared-memory parallel nodes via the Kokkos Node API. As a result, the kernels associated with the Tpetra computational classes are written once, as stateless serial routines, and then fused with the specified Kokkos nodes at compile time. Commonly-used methods – e.g., standard norms, inner products, vector combinations – are provided for computational classes. However, there is a limit to the number of computational primitives that can be provided for a given data class. It is not possible for the library developers to anticipate every operation that an application developer or algorithm designer could require; even if these were enumerated, the maintenance of these by Tpetra developers is not appropriate.

The support for generic shared-memory parallel programming becomes a liability at this point. A developer wishing to extend to the functionality of a Tpetra computational class on a shared-memory node seems to have two choices: utilize an appropriate (perhaps identical) programming model to implement this new functionality; or forgo node-level parallelism, extract the data from the target class and perform the necessary computation, thereby introducing a serial bottleneck.

Fortunately, the Kokkos API that Tpetra uses can be exposed to Tpetra users as well, in order to allow extension of native library capability. Tpetra provides the Tpetra *Reduction/Transformation Interface* (Tpetra::RTI), an API for executing user-provided kernels on Tpetra vectors. Under this approach, a user authors a stateless serial kernel that is compiled and executed on the data associated with one or more Tpetra vectors. This may be contrasted with the standard approach for extending scientific library objects, namely, to grab a pointer to the data and perform some computation. The latter requires the user to bring the data to the (external) code and to write the code efficiently. The Tpetra::RTI approach, however, can be thought of as providing a channel by which the user may send the code to the data (whether it be local or in distinct memory space, such as with accelerators). This adheres to the guiding principle that domain specialists should, whenever possible, be permitted to focus solely on the application code, without having to worry about the idiosyncrasies of shared-memory computing.

The Tpetra::RTI allows the user to provide kernels at nested levels of complexity, where more complex levels require more work on the part of the user but allow for greater descriptive ability. At the finest grain level, the user can author a bonafide Kokkos kernel, as specified by the Kokkos Node API. For example, considering the DAXPY example from Section 2.2, with some cosmetic changes to the DAXPY kernel, we could implement a DAXPY for Tpetra :: Vector objects like follows:[2]

```
RCP<Tpetra::Vector<double> > x=createVector<double>(map);
RCP<Tpetra::Vector<double> > y=createVector<double>(map);
Tpetra :: RTI:: detail :: binary_transform (*y, *x, DAXPY(2.0));
```

For operations consisting of simple, element-wise combinations of the input/output vectors, a function object operating on these values will suffice. These simple functors can be converted, at compile-time via zero-overhead compile-time polymorphism, into Kokkos kernels. Some of these methods are listed in Table 5. The example is modified as so:

```
struct DAXPYfunctor {
  double alpha;
  double operator()(double y, double x){return y+alpha*x;}
};
DAXPYfunctor daxpy; daxpy.alpha = 2.0;
Tpetra :: RTI:: binary_transform ( *y, *x, daxpy );
```

More recently, features to facilitate generic programming in the newly approved C++11 standard have become available. One of these features is lambda expressions/anonymous functions. Using C++11 lambdas, the definition for the functor from the above example can be constructed inline. This makes the above example much more concise, while also having the benefit that the code defining the functor is inline with its usage:

```
Tpetra :: RTI:: binary_transform (
                *y, *x,
                [](double y, double x) {return y + 2.0*x;}
           );
```

While an improvement on the previous example, this is still not optimal; there is redundancy in the provision of the kernel, making the code harder to read and more prone to errors (e.g., the ordering of the arguments must be correct). Therefore, Tpetra::RTI provides some convenience macros to construct these lambdas, call the appropriate Tpetra::RTI method, and

[2]Here we have also utilized the non-member constructor Tpetra :: createVector , which encapsulates a newly allocated Tpetra :: Vector object into a Teuchos RCP smart-pointer, according to a provided Tpetra :: Map object.

Table 5

Selected Tpetra RTI non-member methods

unary_transform(Vector <...> &vec_inout, OP op)
Transform the values of vec_inout using via unary function operator op.
binary_transform (Vector <...> &vec_inout, **const** Vector <...> &vec_in2, OP op)
Transform values of vec_inout using vec_inout, vec_in2 and binary function operator op.
reduce(**const** Vector <...> &vec_in1, **const** Vector <...> &vec_in2, Glob glob)
Reduce values of vec_in1 and vec_in2 using the operators described by glob.
kernelOp(Kernel kernel , **const** RCP<**const** Map<...> > &domainMap, **const** RCP <**const** Map<...> > &rangeMap, **const** RCP<**const** Import<...> > \&importer, **const** RCP<**const** Export<...> > &exporter)
Non-member constructor for a Tpetra :: Operator built from Kokkos kernel object kernel, with optional import/export capability.
binaryOp (Op op, **const** RCP<**const** Map<...> > &domainMap, **const** RCP<**const** Map<...> > &rangeMap, **const** RCP<**const** Import<...> > \&importer, **const** RCP<**const** Export <...> > \&exporter)
Non-member constructor for a Tpetra :: Operator built from binary function object op, with optional import/export capability.

Table 6

Selected Tpetra RTI C++11 lambda macros

TPETRA_UNARY_TRANSFORM(out,expr)
Construct a unary functor based on expression expr and apply it to the vector out.
TPETRA_BINARY_TRANSFORM(out,in,expr)
Construct a binary functor based on expression expr and apply it to the vector out and in.
TPETRA_TERTIARY_TRANSFORM(out,in2,in3,expr)
Construct a tertiary functor based on expression expr and apply it to the vector out, in2 and in3.

pass the appropriate vectors. Some of these macros are listed in Table 6. Using a convenience macro, the example from above becomes:

```
// y = y + 2.0 * x
TPETRA_BINARY_TRANSFORM(y, x, y + 2.0*x );
```

This single line of code, via macro substitution, compile-time polymorphic adapters and template metaprogramming is changed into a loop over the vector data that executes in a hybrid parallel fashion, where the user needed only to write a few characters of code. The significance of this cannot be over-stated. By allowing application and algorithm developers to define new kernels *in-situ* that will execute in parallel, we allow highly parallel, readable and efficient code with kernels designed for a particular algorithm (instead of being adapted from those kernels provided by the library developer).

Tpetra contains an example which implements the conjugate gradient (CG) linear solver, using no native primitives of the Tpetra :: Vector class, where all arithmetic is defined in-situ using the Tpetra::RTI. In addition to demonstrating the usage of Tpetra::RTI, this example demonstrates the benefit of algorithm-specific kernels. In the case of this CG solver, for example, the algorithm can easily be implemented using the library-provided `Tpetra::Vector` primitives. However, the *in-situ* approach allows neighboring primitives to be fused, performing in one pass through a vector what would have otherwise required either multiple passes or the extension of the library with an algorithm-specific kernel. This example illustrates the benefit of easily passing algorithm-specific kernels to the parallel node API, as it allows the fu-

```
template <class S, class LO, class GO, class Node>
void RTICG(const RCP<const Tpetra::Operator<S,LO,GO,Node>> &A,
               RCP<Tpetra::Vector<S,LO,GO,Node>> r)
{
  UNARY_TRANSFORM( x, 0 );                              // x = 0
  S rr = REDUCE( r, r*r, ZeroOp<S>, plus<S>() );        // r'*r
  BINARY_TRANSFORM( p, r, r );                          // p = r
  for (k=0; k<numIters; ++k) {
    A->apply(*p,*Ap);                                   // Ap = A*p
    S pAp = REDUCE2( p, Ap,                             // p'*Ap
                     p*Ap, ZeroOp<S>, plus<S>() );
    const S alpha = rr / pAp;
    BINARY_TRANSFORM( x, p, x + alpha*p );              // x = x + alpha*p
    S rrold = rr;
    rr = BINARY_PRETRANSFORM_REDUCE(
                     r, Ap,                             // fused!
                     r - alpha*Ap,                      // : r - alpha*Ap
                     r*r, ZeroOp<S>, plus<S>() );       // : sum r'*r
    const S beta = rr / rrold;
    BINARY_TRANSFORM( p, r, r + beta*p );               // p = r + beta*p
  }
}
```

Listing 1. A templated CG solver using Tpetra::RTI. TPETRA_macro prefixes have been truncated for brevity.

sion of what would otherwise be multiple independent library calls. This example is contained in the Tpetra source distribution [5]; a concise version appears in Listing 1.

3.3. Multiple and mixed precision computing

One of the benefit of the templated classes in Tpetra is that it supports a wide range of use cases for multiple and mixed precision algorithms. Scientific libraries in C/C++/Fortran that are not based on templates typically employ one of two methods for changing the precision:

(1) a type definition set at configure time that determines the scalar types across the library, or
(2) multiple version of each library method and structure, one for each supported type.

The first approach is utilized by PETSc [10–12]. When configuring and building the PETSc library, a user specifies whether the scalars should be real or complex valued, and whether the underlying field should be **float**, **double**, **long double** or **int**. Choosing 32-bit or 64-bit ordinal types is done in a similar manner. The downside of this approach is that the library is available for only this selection of types. Without making any distinction in the class and method names, it is not even possible to link together multiple diverse builds of the library.

The second approach is utilized, for example, by the ARPACK [21,22] and PRIMME [27,28] eigensolver libraries. This approach typically requires implementing the same algorithm multiple times, once for each desired data type (or combination of data types). The downsides of this are multiple: increased development and maintenance effort on the part of the developer, and the lack of a generic interface for the users. The former of these should not be underestimated. In the case of ARPACK (a Fortran code), most subroutines are implemented four times, for the four Fortran floating types (single, double, complex single and complex double).

The templated approach used in Tpetra requires that Tpetra developers write and maintain a single, generic implementation. C++ template specialization allows distinct versions to be written for particular types, when necessitated by efficiency or analytic considerations (e.g., differences between real and complex implementations of an algorithm). However, our experience in eigensolver and linear solver development in Trilinos suggests that these changes are typically localized to local components of an algorithm.

The decision as to which scalar, ordinal and node types are built into the library is decided at configure time; the default is to build against the standard **double** /**int**, for all enabled Kokkos nodes. However, users can always add support for any additional scalar/ordinal combinations that they need. This includes any nonstandard scalars or ordinals, a capability which is not possible for non-templated libraries without modifications to their source (fortunately, the bulk of these scientific libraries are open source, so this is a possibility).

Another benefit of the templated approach is to support simultaneous usage of different precisions. Code can be written which utilizes Tpetra objects templates on different scalar or ordinal types. This allows the development of algorithms using multiple precisions, the goal being to compute a quantity to a specified precision while exploiting lower precision data and arithmetic in some computations in order to lower the cost of the computation or communication required by the algorithm.

For example, one provision for multi-precision algorithms in Tpetra comes via wrappers for the Tpetra sparse matrix class. The Tpetra class CrsMatrix<Scalar> inherits from Operator<S>; the latter describes a virtual method to apply linear operators to vectors/multivectors:

```
virtual void Tpetra :: Operator<S>::apply(
    const Tpetra :: Vector<S> &x,
    Tpetra :: Vector<S> &y,
    ...
) const;
```

However, the CrsMatrix<S> also contains a templated multiplication method:

```
template <class S2, class S3>
void Tpetra :: CrsMatrix<S>:: localMultiply (
    const Tpetra :: Vector<S2> &x,
    Tpetra :: Vector<S3> &y,
    ...
) const
```

This method is fully generic, allowing a sparse matrix with values of type S to be multiplied by a vector with values of type S2 into a vector with values of type S3. Typically, such methods are not directly used. It is customary to utilize a wrapper, such as the following:

```
template <class SIN, class SOUT>
RCP<Tpetra::Operator<SOUT> >
Tpetra :: createCrsMatrixMultiplyOp(
            RCP<const CrsMatrix<SIN> A);
```

This method accepts a sparse matrix with values of type SIN and wraps it in an operator over the field SOUT. The matrix entries are unchanged, but it allows the matrix to be transparently used in algorithms with vectors of scalar type SOUT. Of course, it must make numerical sense to do so; applying a complex-valued matrix to a real-valued vector will in general not produce a real-valued vector. However, the contrast is reasonable; you can encapsulate a real-valued matrix into a complex-valued operator and utilize an algorithm implemented with complex-valued vectors. Similarly, you could use a single-precision-valued matrix in a double-precision-valued algorithm. This will result in a significant speedup of the sparse matrix–vector multiplications.

Another possibility is the construction of explicit mixed-precision algorithms. While the Tpetra library supports templating on extended precision types, these are computationally expensive. One solution is to employ the extended precision types where necessary, and to fall back on less expensive types where it is numerically acceptable to do so. To demonstrate this, Tpetra contains an example [6] implementing a flexible conjugate gradient method that recursively calls itself for preconditioning. The result is that a linear problem in quad-double precision can be preconditioned by a solver in double–double precision, which can itself be preconditioned by a solver in double precision, and so forth. The result is that the lower precision iterations act as excellent and efficient preconditioners for the more expensive higher-precision iteration. The output of the example is listing in Fig. 1 for the case where a solve over **qd_real** is preconditioned by **dd_real** which is preconditioned by **double**. The solution to 62 digits requires 2.84 s, almost all of this time consumed by the double-precision iteration. This is compared to 13.0 s for **qd_real** preconditioned by **dd_real** and 29.4 s for **qd_real** alone. The algorithm is in Listing 4; note, the bottom level of precision is preconditioned using a diagonal preconditioner and solved to a looser tolerance. The types used are provided by the Tpetra :: TypeStack utility; this is a static linked-list of types, and its usage is illustrated in the algorithm and in the driver in Listing 3. These examples are simple illustrations of what is possible using mixed-precision. Upcoming work in Tpetra will allow for arbitrary scalar types in other phases of computation, as well as bandwidth-saving methods for mixed-precision communication.

3.4. Heterogeneous architecture support

The typical usage of Tpetra falls under the single-program/multiple-data model, where all participating nodes are running the same program. For example,

```
Running test with Node==Kokkos::SerialNode on rank 0/1
  Beginning recursiveFPCG<qd_real>
    Beginning recursiveFPCG<dd_real>
    |res|/|res_0|: 9.877869e-15
    |res|/|res_0|: 5.294857e-28
    |res|/|res_0|: 6.209196e-42
    Convergence detected!
    Leaving recursiveFPCG<dd_real> after 2 iterations.
  |res|/|res_0|: 1.528114e-32
    Beginning recursiveFPCG<dd_real>
    |res|/|res_0|: 1.975142e-13
    |res|/|res_0|: 2.597052e-27
    |res|/|res_0|: 1.352672e-40
    Convergence detected!
    Leaving recursiveFPCG<dd_real> after 2 iterations.
  |res|/|res_0|: 3.512840e-62
  Convergence detected!
  Leaving recursiveFPCG<qd_real> after 1 iterations.
|b - A*x|/|b|: 3.513031e-62
================================================================
Timer Name Global time (num calls)
----------------------------------------------------------------
recursiveFPCG<qd_real> 0.04362 (0)
recursiveFPCG<dd_real> 0.02304 (0)
recursiveFPCG<double> 2.78 (0)
================================================================
```

Figure 1. Recursive linear solve using the implementation from Listing 2.

```
#include <Tpetra_HybridPlatform.hpp>
#include ''MyUserDriver.hpp''
int main(int argc, char *argv[])
{
  // Get the default communicator
  RCP<Comm> comm = ...
  // Read machine file from XML and initialize the HybridPlatform
  RCP<Teuchos::ParameterList> machinePL = Teuchos::parameterList();
  Teuchos::updateParametersFromXmlFile(xmlFileName, machinePL);
  Tpetra::HybridPlatform platform(comm,*machinePL);
  // instantiate a driver object, set any data, have platform run the driver
  MyUserDriver driver (...) ;
  platform.runUserCode(driver);
}
```

Listing 2. An example using the HybridPlatform utility.

they may all be participating in the construction of a Tpetra::Map or the addition of two Tpetra::Vector objects. However, "single program" here means that while the outline of the program with respect to the Tpetra objects is typically the same, the Tpetra objects themselves may be templated on different Kokkos node types. For example, it is fully appropriate for a Tpetra <**double**,**int**, **int**,Kokkos::SerialNode> to be running in parallel (perhaps on MPI rank 0) with a Tpetra<**double**, **int**, **int**,Kokkos::OpenMPNode> (perhaps on MPI rank 1). The communication between them happens from some MPI-capable thread on the CPU and makes no assumption on the node type.

In order to easily manage this capacity for heterogeneity among the nodes, Tpetra provides the utility class Tpetra::HybridPlatform. This class has two significant methods. The first is the constructor:

```
Tpetra::HybridPlatform (RCP<Comm> comm,
                        ParameterList machine_file)
```

```
int main(int argc, char *argv[])
{
  /* ... boilerplate setup ... */
  // create the platform object
  Tpetra::HybridPlatform platform(comm,*machine_file);
  // Define the type stack: qd_real -> dd_real -> double
  TPETRAEXT_TYPESTACK3(MPStack, qd_real, dd_real, double )
  // instantiate a driver on the scalar stack
  MultiPrecDriver<MPStack> driver;
  // run the driver
  platform.runUserCode(driver);
}
```

Listing 3. A driver for the recursive, mixed-precision CG in Listing 4, using the TypeStack and HybridPlatform utilities.

```
template <class TS, class LO, class GO, class Node>
void recursiveFPCG(ParameterList &db)
{
  typedef typename TS::type T;
  typedef typename TS::next::type T2;
  /* ... get vectors/matrices from my database ... */
  for (int k=0; k<numIters; ++k)
  {
    A->apply(*p,*Ap);                                    // Ap = A*p
    T pAp = REDUCE2( p, Ap,
                     p*Ap, ZeroOp<T>, plus<T>() );       // p'*Ap
    const T alpha = zr / pAp;
    BINARY_TRANSFORM( x, p, x + alpha*p );               // x=x+alpha*p
    BINARY_TRANSFORM( rold, r, r );                      // rold=r
    T rr = BINARY_PRETRANSFORM_REDUCE(
                       r, Ap,                            // fused!
                       r - alpha*Ap,                     // r-alpha*Ap
                       r*r, ZeroOp<T>, plus<T>() );      // sum r'*r
    if (TS::bottom) {
    // bottom of the type stack; precondition by diagonal
    TERTIARY_TRANSFORM( z, diag, r, r/diag );            // z=D\r
    }
    else {
      // precondition by recursion
      ParameterList &db_T2 = db.sublist(``child'');
      auto bx_T2 = db_T2.get< RCP<VectorT2>>(``bx'');
      BINARY_TRANSFORM( bx_T2, r, as<T2>(r) );           // b_T2 = (T2)r
      recursiveFPCG<typename TS::next,LO,GO,Node>(db_T2);
      BINARY_TRANSFORM( z, bx_T2, as<T>(bx_T2) );        // z=(T)bx_T2
    }
    const T zoro = zr;
    typedef ZeroOp<pair<T,T>> ZeroPTT;
    auto plusTT = make_pair_op<T,T>(plus<T>());
    pair<T,T> both = REDUCE3( z, r, rold,                // fused!
                        make_pair(z*r, z*rold),          // z'*r
                        ZeroPTT, plusTT );               // z'*r_old
    zr = both.first;
    const T znro = both.second;
    const T beta = (zr - znro) / zoro;
    BINARY_TRANSFORM( p, z, z + beta*p );                // p=z+beta*p
  }
}
```

Listing 4. A recursive, mixed-precision CG solver using Tpetra::RTI. TPETRA_macro prefixes have been truncated for brevity.

where comm is a Teuchos communicator (an abstract class for performing message passing) and machine_file is a database of rules for determining the Kokkos node type for each participating process. The second method accepts a user-authored class:

```
template<class UserCode >
void runUserCode (UserCode &code)
```

The template parameter UserCode is required to have a method run, templated on node type and accepting a Kokkos node and a communicator object. The function of the method Tpetra :: HybridPlatform :: runUserCode is to:

(1) query the communicator object for the process id;
(2) determine the Kokkos node type (e.g., SomeNode) according to the machine file and the process id;
(3) instantiate a Kokkos node of type SomeNode, with parameters from the machine file; and
(4) call the method run<SomeNode>, passing the communicator object and the newly instantiated node.

The benefit of this class is that a Tpetra user is not responsible for allocating nodes, or even knowing which nodes are present in the library. The enumeration of the available nodes occurs inside the HybridPlatform class, as does the allocation of these nodes, according to the machine file. Furthermore, the contents of the machine file are not static; the database can be modified at runtime or read from an XML file. The file takes the following format:

```
<ParameterList>
  <ParameterList name="MATCH">
    <Parameter name="NodeType" type="string"
               value="NODE">
    ... parameters used to construct this node type ...
  </ParameterList>
  ... parameter lists for other node types ...
</ParameterList>
```

where NODE is a string designating one of the supported node and MATCH is a string used to match processor identifiers against node types:

%M=N	matches any rank such that mod(rank, M) is N
[M,N]	matches any rank in the interval $[M, N]$
=N	matches rank N
default	matches if no other node matches

For example, a machine file using Kokkos::OpenMPNode with 8 threads on even ranks and Kokkos::ThrustGPUNode on odd ranks might look like:

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

A side effect is that the main(argc,argv) routine now becomes boilerplate code; the significant portion of the code happens in the run method. See Listing 2 for an example.

3.5. Fine tuning/targeting through template specializations

Another possibility with Tpetra involves the ability to use full or partial specialization to optimize/tune implementations for cherry-picked scenarios. While the hope is that the generic node support in Kokkos will serve Tpetra classes well, some shared-memory nodes are not amenable to this approach. For example, unlike NVIDIA's CUDA, the OpenCL [30] SDK does not support templates or template metaprogramming. Therefore, it is not currently possible to support OpenCL using the current generic approach.

However, it is possible to write specialized versions of the Tpetra classes, *de novo*, templated on OpenCL and the needed data types:

```
class MyOpenCLNode { /* my implementation here */ };
template <> class Tpetra :: Map<int,int,MyOpenCLNode> {
  // implement Map<int,int> using OpenCL
};

template <> class Tpetra :: Vector<double,int, int,
                                   MyOpenCLNode> {
  // implement Vector<double, int, int> using OpenCL
};
  // and so on, for other classes and scalar/ordinal
     combinations that are needed
```

While this example involves a non-trivial amount of work,[3] other classes may be more amenable to modification in this way. One benefit of this specialization approach is that it happens external to the generic implementations in Tpetra. As such, it could be easily added by a vendor or other external developer to support a custom build of Tpetra.

4. Drawbacks

The generic programming techniques utilized in Tpetra are not without their drawbacks. This section lists some of the more significant downsides of this programming paradigm and discusses the attempts to address them in our design.

One of the biggest drawbacks associated with templated code is the cost paid at compile time. Because the definition of a template class is not complete without a specification of all template parameters, it is often the case that large parts of the software are built repeatedly, as they are needed by different compilation units. For example, building a test for the MultiVector class will cause the header files for the Map and MultiVector class to be included, and those classes compiled for all necessary templated parameter combinations. Later, perhaps, building a test of the CrsMatrix class will cause these same headers to be included again, and their contents built again. This replicated effort can cause a significant amount of time to be spent in the compilation phase. Furthermore, it may be the case that large portions of the library code have to be read and compiled to create a single executable; this can consume a large amount of resources, and this has been the cause of more than a few failures for less-robust compilers over the years. One solution is to use explicit instantiation. Using this approach, a list of all instantiations of the template classes is created, put into a source file, and then built. Furthermore, these can be spread across multiple source files, so that a multi-core machine can be occupied by building multiple files at once. The downside of this approach is determining, in advance, the myriad of combinations that need to be built. This issue is further exacerbated when user-defined ordinal and scalar types are considered. Therefore, whatever support utilities (typically, explicit instantiation macros) must be made available to the user as well. The benefit of explicit instantiation is that, if it is possible to implement correctly, the build mechanics become similar to that of non-templated libraries.

[3]This is why it has not been done yet.

Another downside of compile-time polymorphism is the size of the executable. In C++, all routines that could possibly be called will be built and linked into an executable; this requires extended time to build and link, and potentially generates a large executable. This applies to the instantiations (implicit or explicit) of a template class. In a certain sense, this cost is a direct reflection of the benefit of templated code. For example, the privilege of computing with both Vector<**float**> and Vector<**double**> requires that both have been built and linked into the executable. Still, this short-coming is a consequence of compile-time polymorphism, and is not necessary present in interpreted languages such as Python and Matlab.

The single biggest risk for this approach concerns the quality of the code produced by the compiler. Many of the template metaprogramming use cases described here rely on appropriate inlining by the compiler. This is especially critical for the fusion of user kernels and parallel skeletons in the Tpetra::RTI and Kokkos Node APIs. The inability or unwillingness of the compiler to perform this optimization will result in code that may be unacceptably slower than hand-coded implementations. Unfortunately, the use of high-level languages and sophisticated language constructs ultimately rests on the availability of quality compilers.

5. Conclusion

This paper illustrated some of the benefits of generic programming to applications in scientific computing, as illustrated via the Tpetra package. This paper does not cover the whole of this package; other functionality, such as block entry vectors and matrices, is present, in addition to utility code supporting advanced communication and file I/O. Further development will expand on the generic programming capability, especially as concerns support for mixed-precision communication and algorithms.

Acknowledgements

We would like to acknowledge the efforts of contributors to the Tpetra, Teuchos and Kokkos packages that have supported this work: Roscoe Bartlett (ORNL), Alan Williams (SNL), Carter Edwards (SNL), Radu Popescu (EPFL) and Paul Sexton. We also appreciate the suggestions of the anonymous reviewers.

References

[1] D.H. Bailey, High-precision software directory, http://crd.lbl.gov/~dhbailey/mpdist/, 2011.

[2] D.H. Bailey, Y. Hida, X.S. Li and O. Thompson, ARPREC: an arbitrary precision computation package, LBNL Technical Report 53651, 2002.

[3] C.G. Baker, Developing large-scale scientific software for generic multi-core nodes, in: *SIAM Conference on Parallel Processing for Scientific Computing*, February 2010, available at: http://info.ornl.gov/sites/publications/Files/Pub23325.pptx.

[4] C.G. Baker, Parallel programming with Kokkos and Tpetra, Trilinos User Group 2010, November 2010, available at: http://info.ornl.gov/sites/publications/Files/Pub27346.pdf.

[5] C.G. Baker, Tpetra conjugate gradient RTI example, available at: http://trilinos.sandia.gov/packages/docs/dev/packages/tpetra/doc/html/RTIInlineCG_8hpp_source.html or source file Trilinos/packages/tpetra/example/RTInterface/RTIInlineCG.hpp, 2011.

[6] C.G. Baker, Tpetra recursive mixed-precision conjugate gradient example, file:///Users/ogb/Trilinos/packages/tpetra/doc/html/MultiPrecCG_8hpp_source.html or source file Trilinos/-packages/tpetra/example/MultiPrec/MultiPrecCG.hpp, 2011.

[7] C.G. Baker, H.C. Edwards, M.A. Heroux and A.B. Williams, An abstract model for programming multi-core nodes, in: *SIAM Annual Meeting*, July 2009 (contact author for slides).

[8] C.G. Baker, H.C. Edwards, M.A. Heroux and A.B. Williams, A light-weight API for multicore programming, in: *Proc. 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, February 2010, available at: http://info.ornl.gov/sites/publications/Files/Pub23109.pdf.

[9] C.G. Baker, U.L. Hetmaniuk, R.B. Lehoucq and H.K. Thornquist, Anasazi software for the numerical solution of large-scale eigenvalue problems, *ACM Trans. Math. Softw.* **36** (2009), 13:1–13:23.

[10] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith and H. Zhang, PETSc users manual, Technical Report ANL-95/11, Revision 3.1, Argonne National Laboratory, 2010.

[11] S. Balay, J. Brown, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith and H. Zhang, PETSc homepage, http://www.mcs.anl.gov/petsc, 2011.

[12] S. Balay, W.D. Gropp, L.C. McInnes and B.F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen, eds, Birkhäuser, New York, 1997, pp. 163–202.

[13] R. Bartlett, Teuchos C++ memory management classes, idioms, and related topic, SNL Technical Report SAND2010-2234, 2010.

[14] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, M. Heroux and R. Preis, Ldrd report: parallel repartitioning for optimal solver performance, SNL Technical Report SAND2004-0365, 2004.

[15] T.A. Davis and Y. Hu, The University of Florida sparse matrix collection, available at: http://www.cise.ufl.edu/research/sparse/matrices.

[16] Epetra 10.6 release notes, http://trilinos.sandia.gov/changelog-10.6.html, 2010.

[17] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams and K.S. Stanley, An overview of the trilinos project, *ACM Trans. Math. Softw.* **31** (2005), 397–423.

[18] Y. Hida, X.S. Li and D.H. Bailey, Algorithms for quad-double precision floating point arithmetic, in: *Proc. 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, Washington, DC, USA, 2001, pp. 155–162.

[19] J. Hoberock and N. Bell, Thrust: a parallel template library, Version 1.3.0, available at: http://www.meganewtons.com, 2010.

[20] Intel Corporation, Intel threading building blocks homepage, http://www.threadingbuildingblocks.org, 2006.

[21] R.B. Lehoucq, D.C. Sorensen and C. Yang, *Arpack Users' Guide*, Soc. Industr. Appl. Math., Philadephia, PA, 1998.

[22] R.B. Lehoucq, D.C. Sorensen and C. Yang, ARPACK homepage, http://www.caam.rice.edu/software/ARPACK, 2011.

[23] NVIDIA Corporation, NVIDIA CUDA homepage, http://www.nvidia.com/cuda, 2009.

[24] OpenMP ARB, OpenMP official homepage, http://openmp.org/wp, 1997.

[25] E.T. Phipps, R.A. Bartlett, D.M. Gay and R.J. Hoekstra, Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation, in: *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bcker, P. Hovland, U. Naumann and J. Utke, eds, Lecture Notes in Computational Science and Engineering, Vol. 64, Springer, Berlin, 2008, pp. 351–362.

[26] J. Reinders, *Intel Threading Building Blocks*, 1st edn, O'Reilly & Associates, Sebastopol, CA, 2007.

[27] A. Stathopoulos and J.R. McCombs, PRIMME: preconditioned iterative multimethod eigensolver-methods and software description, *ACM Trans. Math. Softw.* **37** (2010), 21:1–21:30.

[28] A. Stathopoulos and J.R. McCombs, PRIMME homepage, http://www.cs.wm.edu/~andreas/software, 2011.

[29] B. Stroustrup, *The C++ Programming Language: Special Edition*, 3rd edn, Addison-Wesley, Reading, MA, 2000.

[30] The Khronos Group, OpenCL overview, available at: http://www.khronos.org/opencl, 2008.

[31] The Open Group, <pthread.h>, http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html, 1997.

[32] Trilinos homepage, http://trilinos.org, 2012.

[33] Trilinos packages list, http://trilinos.sandia.gov/packages/.

[34] Wikipedia, C++ classes – Wikipedia, the free encyclopedia, available at: http://en.wikipedia.org/w/index.php?title=C%2B%2B_classes, 2012 (online; accessed 18 April 2012).

[35] Wikipedia, Template (C++) – Wikipedia, the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Template_(C%2B%2B) (online; accessed 18 April 2012).