Scientific Programming 20 (2012) 223–239
DOI 10.3233/SPR-2012-0344
IOS Press

Design considerations for a flexible multigrid preconditioning library

Jérémie Gaidamour [a,*], Jonathan Hu [b], Chris Siefert [a] and Ray Tuminaro [b]
[a] *Sandia National Laboratories[**], Albuquerque, NM, USA*
[b] *Sandia National Laboratories, Livermore, CA, USA*

Abstract. MueLu is a library within the Trilinos software project [An overview of Trilinos, Technical Report SAND2003-2927, Sandia National Laboratories, 2003] and provides a framework for parallel multigrid preconditioning methods for large sparse linear systems. While providing efficient implementations of modern multigrid methods based on smoothed aggregation and energy minimization concepts, MueLu is designed to be customized and extended. This article gives an overview of design considerations for the MueLu package: user interfaces, internal design, data management, usage of modern software constructs, leveraging Trilinos capabilities, linear algebra operations and advanced application.

Keywords: Multigrid, algebraic multigrid, AMG, smoothed aggregation, energy minimization, MueLu, Trilinos, software design

1. Introduction

The repeated solution of sparse linear systems is often a key computational bottleneck for large-scale computer simulations, and multigrid techniques [6] are often employed precisely because of their algorithmic and parallel scalability. The basic idea of multigrid is to use the representation of the same problem at different scales to accelerate the resolution of the finest discretized problem. Multigrid actually involves two complementary processes, *smoothing* and *coarse grid correction*. The smoothing generally consists of applying a relaxation method like Jacobi or Gauss–Seidel to reduce certain errors on the fine grid discretization. The error that has not been eliminated by the smoothing process must be accurately represented on a coarser grid. The aim of the coarse grid correction is then to eliminate this error. This can be done by applying recursively the two-grid algorithm to this new grid. In geometric multigrid, the knowledge of the problem geometry is used to create discretization of the problem at different scales, and the fine and coarse grid errors can be interpreted as high and low frequency. In algebraic multigrid (AMG), the hierarchy of grids is derived directly from the finest grid by using graph algorithms, and the fine and coarse grid errors can be characterized as high and low energy, respectively. In both methods, a restriction operator is used to transfer the residual to the next coarser grid, and a prolongation operator interpolates it back to the fine grid after the coarse grid correction.

Over the years, there have been successful software efforts to provide open source high performance parallel multigrid libraries:

- The HYPRE project [8,9] provides the BoomerAMG package [12] that is a parallel implementation in C of classic Ruge–Stueben (F/C) algebraic multigrid with several coarsening and relaxation schemes. HYPRE also provides various structured multigrid solvers and Maxwell solvers. Interfaces to Fortran, C++, Python and Java are available.
- ML [11] is currently the main multigrid preconditioning package of the Trilinos Project [17–20,24]. ML primarily focuses on smoothed aggregation [26] based preconditioners. ML contains a variety of parallel multigrid schemes like standard smoothed aggregation, Petrov–Galerkin for nonsymmetric systems, and two schemes for Maxwell's equations. ML is written in C. Additionally, ML has Python and C++ interfaces [23]

*Corresponding author: Jérémie Gaidamour, Sandia National Laboratories, MS 1320, P.O. Box 5800, Albuquerque, NM 87185-1320, USA. E-mail: jngaida@sandia.gov.

**Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

and can interoperate with some other Trilinos packages.

As part of the Trilinos project, we are developing a new object-oriented multigrid solver named MueLu. MueLu is meant to be a replacement for ML in existing applications, a research vehicle, as well as the delivery vehicle for future algorithms. Using multigrid as a blackbox is often not enough for challenging problems arising from multi-physics simulations that may have boundary layers, mesh stretching, material discontinuities, or other features that stress linear solvers. A highly versatile framework is needed to deploy customizable MG methods as well as support future research. Both the software design and the algorithms must be versatile: implementation of problem-specific improvements directly in the solver has to be straightforward from a software standpoint, but algorithms should also be easy to reuse and adaptable to other problems. A specific goal of the MueLu design is to support recently developed multigrid approaches based on energy minimization [27]. These methods allow more flexibility in the choice of inter-grid transfer operators, which in turn allows better control over the quality and the application cost of the coarse grid correction. These methods extend the applicability of AMG methods to challenging problems like those arising from systems of PDEs. In addition, such algorithms can be viewed as a generalization of other successful multigrid algorithms, like the smoothed aggregation approach used in ML. MueLu is primarily intended to make such new multigrid approaches available. While algorithmic flexibility is attractive, the software framework must make this easily achievable.

Therefore, the primary design goal of our project is to create a highly configurable multigrid framework leveraging the flexibility of these new multigrid approaches. Control over all the parameters of the algorithms must be available via the high-level interface. We want to enable advanced users to customize deeply all the components of a multigrid solver in an easy way. Additionally, special attention was paid to the code design during development to provide straight-forward extensibility. In particular, the modularity of MueLu allows the reuse and reorganization of existing algorithm components to help build other multigrid methods. The intentional design of MueLu also allows for a variety of multigrid methods like geometric, classic and aggregation based methods; MueLu is not limited to energy-minimization based methods. In particular, MueLu is designed to support the key algorithms in ML that are in use today: standard smoothed aggregation, Petrov–Galerkin aggregation for nonsymmetric systems and Maxwell solvers.

Finally, MueLu must also be robust enough for everyday use on a variety of platforms. MueLu is written in C++ and uses modern programming concepts, both of which facilitate the development, maintenance and extension of the code. Performance and scalability are also important design requirements of MueLu, but these objectives are mainly achieved by leveraging some of the newest Trilinos capabilities. The Trilinos framework also allows outsourcing of some of the multigrid algorithm components and also provides tools to improve the maintainability of our software.

This paper is organized as follows. First, we give an overview of the user interface in Section 2. In Section 3, we describe some details of the internal design. In Section 4 we explain how we leverage the capabilities of Trilinos packages. In Section 5, we describe some advanced applications that demonstrate the flexibility of our new framework. Finally, we make some concluding remarks in Section 6.

2. User interfaces

In this section, we first introduce some design concepts that help to explain the MueLu interfaces. We then present two types of interfaces: the native interface in which the user interacts fully with MueLu objects, and simplified interfaces that hide much of the details but extend naturally to the native interface.

2.1. General design philosophy

A main theme of the MueLu design is to allow access to all parts of the multigrid setup and solution phase, so that existing multigrid methods can be tuned for optimal performance and so that new algorithms can be developed and tested without undue burden. From the point of view of software design, the multigrid setup phase mainly consists of the creation of new objects that will be used during the solution phase. A hierarchy of coarse grid problems is created that will later be used during the multigrid solution phase. Examples of persistent objects produced during the setup phase are prolongation, restriction and coarse matrices. Multigrid setup also generally requires many temporary objects, such as graphs, aggregates and tentative prolongators. The specifics of how the persistent and temporary objects are created depends on the particular

```
for (int i=1; i<nLevels; i++) {
  P = prolongatorFact.Build(...);
  R = restrictionFact.Build(...);
  Ac = projectionFact.Build(...);
}
```

Fig. 1. Internal MueLu multigrid setup phase illustrating correlation of algorithmic components and factories.

multigrid method. Because of this required generality, we leverage certain software design concepts.

MueLu makes extensive use of the *factory design pattern* [10] for the creation of hierarchy objects. Basically, factories are objects intended to ease the creation of other objects. For example, a prolongation factory provides a method allowing repeated creation of prolongation matrices. Behavior of factories can be guided by parameters supplied as input via either factory constructors or "set" methods. A factory can also be easily replaced by another factory implementing the same interface, thus permitting one mechanism for creating objects to be replaced by another.

To illustrate the use of factories in MueLu, we show some excerpted internal code in Fig. 1. We emphasize that this particular example is internal to MueLu, and that the user is not required to write such a loop. There is a strong correlation between factories and algorithmic blocks of the multigrid setup phase. The exact definition of `Build` methods is discussed in Section 3.

We expect that virtually all users will interact with MueLu through the simplified interface. End users can choose to ignore the internal design and just specify the behavior of the solver using a monolithic list of parameters. In reality, however, a MueLu preconditioner is not a single object, but rather the end product of the interplay of many individual objects (factories). These factories correspond to multigrid algorithmic components. So while a user may not think of these objects as factories, understanding that a variety of objects are combined together is an important concept. Design patterns are generally only invoked for internal design consideration, but in MueLu they are also necessary to fully understand the interface.

2.2. Native interface

The native interface uses the factories to directly manipulate the multigrid algorithm components. We have designed the native interface to facilitate the development and implementation of new advanced algorithms. MueLu provides factories such that all parts of the preconditioner can be explicitly customized. This modularity serves several purposes:

(1) The design is flexible, such that pieces of the algorithm can be swapped in and out.
(2) Advanced preconditioners can be created that are tailored to specific problems.
(3) The design is extensible, thus allowing for future algorithm development.

For example, the user has control over coarsening, detection of strength of coupling between unknowns, grid transfer sparsity patterns, coarse near nullspace representation, initial guess of the transfer operator, etc., in the high-level interface. In addition, new factories can be implemented and used without modifying any core components of MueLu.

To use this interface, the user specifies how to build a level of the hierarchy by defining a set of factories. During the setup phase, these factories work together to build the entire multigrid hierarchy. The user perspective follows:

(1) The user instantiates the factories that must be used. This implicitly defines the type of multigrid method. Each factory's behavior is guided by options given to its constructor. These options are a few local parameters and/or any factories provided as input. At no point does the user have to provide a single monolithic parameter list that describes the entire multigrid preconditioner.
(2) Multigrid levels are automatically built by the user-provided factories.
(3) Level data are stored in an instance of the class `Hierarchy`.
(4) The method `Iterate` of `Hierarchy` allows the application of multiple multigrid cycles. The `Hierarchy` object can also be used as a preconditioner of an iterative method.

`Hierarchy` is the central class of our software. It implements both the algorithms of the setup phase (item 2) and of the solution phase (item 4). `Hierarchy` also stores the information needed to apply the multigrid methods (item 3).

A typical usage of the native interface is presented in Fig. 2. In this example, the user specifies the use of a smoothed aggregation factory to build prolongators. The linear problem is represented by using vector and matrix classes provided by TPETRA [13], a linear algebra package of Trilinos. The interface of MueLu accepts either EPETRA [15] or TPETRA matrices. This is discussed further in Section 4.2. MueLu uses TEUCHOS reference counted pointers [3,4,16], or *RCPs*, in interfaces. A reference counted pointer is a type of smart pointer that automatically tracks all the copies of

```
using Teuchos::Comm; // communicator
using Teuchos::RCP; using Teuchos::rcp; // smart pointers
using Tpetra::Map; using Tpetra::Vector; // linear algebra library
using namespace MueLu;

[...] // variable declarations are skipped

// Partition of distributed objects
numElements = 16;
comm = Teuchos::DefaultComm<int>::getComm();
map = Tpetra::createUniformContigMap(numElements, comm);

// Linear problem
A = ...;

// Multigrid hierarchy
Hierarchy H(A);

// Define how to build coarse levels
prolongationFact = rcp(new SaPFactory<double>());

H.Populate(prolongationFact); // build coarse levels

// Setup smoothers
smoother = rcp(new Ifpack2Smoother("point relaxation"));
H.SetSmoother(smoother);

// Solve Ax=b
X = Tpetra::createVector(map); X->randomize();
B = Tpetra::createVector(map);
H.Iterate(*B,30,*X); // 30 iterations
```

Fig. 2. Example illustrating the native interface of MueLu. This example creates a smoothed aggregation multigrid method.

the object pointer – the memory is freed when it is no longer referenced by any object. Using such pointers at the interface level avoids problems of data ownership transfer between applications and libraries. RCPs are also used internally in MueLu to reduce the danger of memory leaks (especially in the case of exception handling).

A more advanced usage is presented in Fig. 3. The user now fully specifies how to create the multigrid hierarchy by providing a prolongation, a restriction and a projection factory. Smoothers and coarse solvers are also defined explicitly. Separately specifying the restriction operators is useful for nonsymmetric problems like convection–diffusion systems. The user is free to modify the formation of the coarse grid operator by deriving his own factory from the default projection factory. In this way, the coarse grid representation of auxiliary information such as coordinates can be explicitly controlled. This example also shows how the smoothed aggregation prolongation factory leans on the capabilities of two other factories: a tentative prolongator factory and an aggregation factory. Here, the user requests a particular aggregation algorithm to be used for building the tentative prolongator. The flexibility imparted by the use of chains of factories allows the user to created tailored preconditioners. Multigrid methods frequently exploit the knowledge of the matrix nullspace to build coarse grid corrections. Such additional information can be added directly to the hierarchy data structure. More details on the Hierarchy `Set()` method are given in Section 3.

We note that in both examples, there is no overarching parameter list that describes the entire preconditioner construction process. Rather, each factory's behavior is entirely specified by a small set of local parameters and the upstream factories specified at its construction time.

Defining more complex MG methods is also possible. Figure 4 demonstrates how hybrid methods can be potentially created by combining geometric multigrid and algebraic multigrid. The multigrid methods are specified by the choice of prolongator factory. The calls to `Populate()` specify the particular levels at which the two different prolongator types have to be

```
// The following header files define Scalar=double and Ordinal=int
// by default and allow us to skip template parameters
#include "MueLu_UseDefaultTypes.hpp"
#include "MueLu_UseShortNames.hpp"

using Teuchos::rcp;
using namespace MueLu;

[...] // variable declarations are skipped

// Multigrid hierarchy (level 0 == fine grid)
Hierarchy H(A);
H.GetLevel(0)->Set("Nullspace", nullspace);

// Configure the multigrid method

// 1- Graph coarsening
aggregationFact = rcp(new UCAggregationFactory());

// 2- Tentative and Smoothed prolongator factories
tentativePFact = rcp(new TentativePFactory(aggregationFact));
prolongationFact = rcp(new SaPFactory(tentativePFact));

// 3- R = P^T and Ac = RAP
restrictionFact = rcp(new TransPFactory());
projectionFact = rcp(new RAPFactory());
projectionFact->setVerbLevel(Teuchos::VERB_HIGH);

// 4- Smoother
smoother = rcp(new Ifpack2Smoother("point relaxation"));
smoother->SetNumIts(2);
smoother->SetBackwardSweep(true);

// 5- Solver used on the coarse grid
coarseSolver = rcp(new Amesos2Smoother());

// Populate Hierarchy
H.SetMaxNumLevel(5);
H.Populate(prolongationFact, restrictionFact, projectionFact);
H.SetSmoother(smoother);
H.SetCoarsestSolver(coarseSolver);
```

Fig. 3. Example illustrating how the interface allows to configure deeply the smoothed aggregation multigrid method.

used. Smoothers are also set for individual levels in the same way.

Finally, the flexibility of the framework also optionally allows one to preserve data between successive setup phases. For example, aggregates formed during coarsening might be restricted to remain the same from one time step to the next. This type of reuse can help reduce setup overhead. The native interface allows this quite easily. Indeed, any hierarchy data can be kept at a user's request during the first run. The user can replace any factory by another that does nothing other than checking that data are already present in the hierarchy.

In summary, the native interface has the following benefits:

(1) There is no need to have complex lists of parameters. Rather, the multigrid preconditioner construction process is controlled by the types of factories that the user creates.
(2) Changing any aspect of the multigrid method is as easy as swapping one object for another.
(3) Adding new algorithms is straightforward. There is no need to modify any existing code. Instead, a new class can be derived and used.

2.3. Simplified interfaces

MueLu provides extensions of the native interface that may be easier to use for end-users. Based on prior

```
// Multigrid hierarchy
Hierarchy H(A);

// Level 0
smoother0 = rcp(new Ifpack2Smoother("point relaxation"));
H.SetSmoother(smoother0, 0);

// Level 1
prolongationFact1 = rcp(new myGeoPFactory());
smoother1 = rcp(new Ifpack2Smoother("point relaxation"));

H.Populate(prolongationFact1, 1);
H.SetSmoother(smoother1, 1);

// Level 2
prolongationFact2 = rcp(new SaPFactory());
smoother2 = rcp(new Ifpack2Smoother("ILUT"));

H.Populate(prolongationFact2, 2);
H.SetSmoother(smoother2, 2);
```

Fig. 4. Example illustrating how factories can be used to create a more advanced multigrid method. A hybrid preconditioner combining geometric and algebraic multigrid is built and smoothers are specified on individual levels.

experience with the ML package, we anticipate that such interfaces will see heavy use, especially with new users.

The first interface makes use of the *facade* design pattern [10]. A facade essentially establishes default values for certain preconditioner options. In MueLu, there is a single facade for each particular problem type, and the naming convention is suggestive of that problem. For each problem type, reasonable default parameters have been pre-determined by the MueLu developers. Problem types includes recommended settings for Poisson, linear elasticity, convection–diffusion, and Maxwell (i.e., H(curl)) problems. Others can be added as needed. MueLu facade classes are derived from the `Hierarchy` class. For example, the constructor of the linear elasticity facade can be called like this: `ElasticityHierarchy(A,nullspace)`. A user still has access to the native interface and control over the preconditioner construction process if wanted. But if such a capability is not used, default factories will be automatically instantiated during the setup phase according to the problem type.

The second simplified interface is driven via parameter lists. Parameter lists are simply lists of key-value pairs. Parameters can be added and retrieved with accessor functions. These parameters can be any data type which uses value semantics (e.g., double, float, int, *double, *float, *int, ...). These parameters can also be pointers to vectors or functions. Parameters may also be other parameter lists, allowing for a hierarchy of lists. Many Trilinos packages can be configured using such a list, and many applications use such a capability to interact with Trilinos packages. Parameter lists are popular thanks to their dynamic aspect. Parameter lists can be loaded from an XML file, and modifications do not require any recompilation. In addition, it is easy to give access to all the configuration options of subpackages in a complex application. An example of usage for MueLu is shown in Fig. 5. Parameter lists can fully describe a multigrid method.

Simplified interfaces are both easy to use and fairly close to the native interface. It is easy to switch from one of the simplified interfaces to the native one by looking at the implementation of the facades or at the parameter list interpreter. In addition, parameter lists can be used in combination with facades and vice versa: default parameters from a facade can be used in a parameter list (using `paramList.set("problem", "elasticity")`), and facades can interpret parameter lists to modify their default behavior as shown in Fig. 6.

3. Internal design

In this section we discuss the internal design of MueLu. We begin with a description of the hierarchy data structure and explain how the hierarchy is populated during the setup phase. In a second part, we

```
// Parameters
Teuchos::ParameterList paramList;

paramList.set("max levels",2);
paramList.set("aggregation: type", "Uncoupled");
paramList.set("aggregation: damping factor", 0.0);

paramList.set("smoother: type","symmetric Gauss-Seidel");
paramList.set("smoother: sweeps",1);
paramList.set("smoother: damping factor",1.0);
paramList.set("smoother: pre or post", "both");

paramList.set("coarse: type","Amesos-KLU");
paramList.set("coarse: max size",32);

// Multigrid hierarchy
Hierarchy H(A);

// Setup
ParameterListInterpreter mueLuFactory(paramList);
mueLuFactory.SetupHierarchy(H);

// Solve Ax=b
H.Iterate(*B,30,*X);
```

Fig. 5. Using MueLu with `Teuchos::ParameterList`.

```
// Linear problem
RCP< Tpetra::CrsMatrix<double> > A = ...;
RCP< Tpetra::Vector<double> > nullspace = Tpetra::
    createVector(map);
nullspace->putScalar(1.0);

// Parameters
Teuchos::ParameterList paramList;
paramList.set("smoother: type", "symmetric Gauss-Seidel");
paramList.set("smoother: sweeps", 4);
paramList.set("smoother: damping factor", 1.0);

// Multigrid hierarchy
ElasticityHierarchy H(A, nullspace, paramList);

// Solve Ax=b
H.Iterate(*B,30,*X);
```

Fig. 6. Using MueLu with both the facade interface and a parameter list.

describe the modular design of the transfer operator construction. We explain how prolongator algorithms can be subdivided and discuss how data transfers between submodules are managed. Finally, we discuss some particulars of the smoothing process.

3.1. The multigrid hierarchy

The central class of the MueLu design is the `Hierarchy`. It represents the grid hierarchy of the multigrid method. `Hierarchy` objects are manipulated directly by the user as described in Section 2 for both configuring multigrid methods (using `Populate()` and `SetSmoothers()`) and applying multigrid cycles (using the method `Iterate()`). The grid hierarchy data are built during the setup phase using the algorithm described in Fig. 1. Basically, this algorithm consists of calling the `Build()` methods of a set of factories for each level of the grid hierarchy. Produced or constructed data are stored in the `Hierarchy` ob-

ject. After this setup phase, this data can be used in the solve step for applying multigrid cycles. Internally, `Hierarchy` is an array of `Level` objects, each object representing one level of the multigrid hierarchy.

Input and output data of factories involved in the setup phase depend on the multigrid methods. For instance:

- The prolongator factory of smoothed aggregation computes both the prolongator matrix and the nullspace of the coarse grid. The coarse nullspace has to be stored for the next recursion step because it is also an input argument of the prolongator factory.
- The prolongator factory of geometric multigrid methods needs geometric information as input and this information must be projected to coarse levels.

Such input and output arguments do not necessarily exist for other multigrid methods. To allow flexibility in how factories operate, the factory `Build()` methods take `Level` objects as input/output arguments. Factory `Build()` methods have then the following prototypes: `Build(FineLevel, CoarseLevel)`. Factories can directly access any data from the levels and also directly populate them.

The `Level` class is based on an associative array design. Through the use of a generic type, the `Level` class allows great flexibility in what type of data can be stored and can thus be used to store the input and output of the factories involved in the construction of the multigrid hierarchy.

Thanks to the uniformity of the factory interfaces, the multigrid setup algorithm is generic. In addition, the generality of the data structures allows them to be reused for any multigrid method.

3.2. Coarse grid correction

One of the key family of classes in MueLu is the transfer class family. These classes in fact determine the type of multigrid method that the `Hierarchy` constructs, e.g., geometric, smoothed aggregation, Ruge Stueben, etc. The purpose of each transfer factory is to produce a particular type of grid transfer matrix. Each transfer factory is highly customizable, and two transfer factories generally share some common algorithm parts. To be able to modify factory configuration and also to reuse algorithm parts of transfer factories, transfer classes have been designed to be highly modular. In particular, each transfer algorithm has a number of logical components, each of which has a corresponding factory. Altering a single phase of the transfer algorithm amounts to supplying a new input factory. This facilitates algorithm development, as developers can experiment with individual components. Additionally, algorithms can be tailored for particular problems, e.g., problems requiring a detection scheme for abrupt material jumps. Although the transfer factories can be customized, the user does not need to know the implementation details. If required input parameters are not supplied, a transfer factory will invoke default factories to generate these parameters. The following section introduces the design of the smoothed aggregation transfer operator as an example.

3.2.1. Example – implementation of the smoothed aggregation prolongator class

In the smoothed aggregation algebraic multigrid method, the coarse grid is defined by using an aggregation algorithm that clusters the nodes of the matrix graph. A first prolongator (called tentative prolongator) is then built. Its coefficients are chosen in order to preserve the zero energy modes of the initial problem on the coarse grid. The quality of the interpolation operator is then improved by using a smoothing procedure (classically a damped Jacobi) that reduces the energy of the coarse basis functions while preserving the correct interpolation of zero energy modes. The steps for building the prolongator of the smoothed aggregation multigrid method follow.

- Build the coalesced graph of the matrix (input: matrix; output: graph).
- Build the aggregates (input: graph; output: aggregates).
- Build the tentative prolongator (input: nullspace, aggregates, matrix; output: tentative prolongator matrix, coarse nullspace).
- Smooth the tentative prolongator (input: tentative prolongator matrix; output: P).

As our goal is to be able to change any part of the algorithm easily (e.g., changing the aggregation algorithm), the algorithm is split into four factories: `GraphFactory`, `AggregationFactory`, `TentativePFactory` and `SaPFactory`. This modularity also allows one to reuse parts of this method within other multigrid methods like energy-minimization. Figure 7 shows the dependency graph of the factories involved in the construction of the prolongator of smoothed aggregation AMG.

A user who wants to use the `SaPFactory` does not have to know the implementation details of the

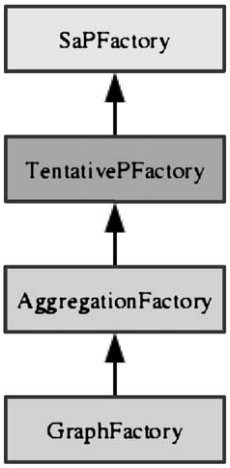

Fig. 7. Dependency graph of the factories involved in the construction of the prolongator of a smoothed aggregation multigrid method.

smoothed aggregation method. In particular, it is not necessary to know which submodules are involved in the process. Default submodules are directly invoked if they are not specified manually. Default factories called in this process can always be changed by advanced users, however. For example, the aggregation algorithm can be changed using `myTentativePFact.SetAggregationFactory(myAggregationFact)`.

3.2.2. Managing data flow between factories

The modular design allows great flexibility but introduces the challenging problems of communication between modules. The data generation and use patterns can be quite varied for two different multigrid methods, and the main challenge is creating a flexible framework that allows for different data flow patterns. One should notice that it is not unusual for a factory to require data produced elsewhere. In fact, the typical case is that the factory does not actually know where the data was created, only that it exists.

`Level` objects store data needed during the solve phase (like prolongators, coarse grid matrices and smoothers) but also allow great flexibility in what type of data can be stored, as seen in Section 3.1. `Level` can thus also act as a "scratch pad" to store temporary data of the setup phase (graph, aggregates, tentative prolongator, coarse nullspace). This is convenient for communicating data from one factory to another. For instance, aggregates generated by the `AggregationFactory` (used in a smoothed aggregation method) can be stored in the `Level` data structure and can be accessed later by any factory. Using this mechanism, a factory does not have to know which factories will use its output in the rest of the algorithm. Each factory is fairly independent and has no information on how it will be linked with other factories. If a sub-module is used twice on the prolongator construction algorithm, the computation will only be done once.

Inserting and accessing data in the `Level` is straightforward:

level.Set("A", Adata, someFact): Object `Adata` is saved in `level` with key "A". The factory `someFact` that created the data is also recorded because several factories may produce data with the same name. For instance, both TentativePFactory and SaPFactory produce a prolongator matrix named "P". The keys of the associative array `Level` are thus the pair ("A", someFact).

level.Get("A", Adata, someFact): Object `Adata` created by the factory `someFact` is retrieved. If this object does not exist, the build method of `someFact` has to be called to generate the data.

Temporary data of `Level` are deallocated as soon as possible via a counter mechanism that works as follows:

- Before the setup phase, factories declare their input data (and increment the corresponding counters). That is, the factories register which data they expect to find in `Level`.
- At the end of the execution of a specific factory's `Build` method, the counter is decremented.
- When the counter reaches zero, temporary data are automatically freed.

Note that if some output of factories is never requested by any other factory, then this output is never stored.

Figure 8 gives an idea of the implementation of `SaPFactory::Build()`.

3.3. Smoothing procedure

In MueLu, smoothers are represented by classes that implement the interface `SmootherBase`. An example of a smoother class is presented Fig. 9. Smoothers are invoked during the solve step using the `Apply()` method.

Even if the same smoothing procedure is applied on each grid of the hierarchy, a separate instance of the smoother class is created for each level. Each smoother object is associated with only one level because preprocessing is generally required before applying a smoother. For example incomplete factors have to

```
void Build(Level &fineLevel, Level &coarseLevel) const {
  // input
  RCP<Operator> Ptent = fineLevel.Get('P', tentativePFact_);
  fineLevel.Release('P', tentativePFact_); // free

  // prolongator smoothing (pseudo-code)
  P = (I - w D^-1 A) Ptent;

  // output
  fineLevel.Set('P', P, this);
}
```

Fig. 8. Implementation of the `SaPFactory::Build()` method.

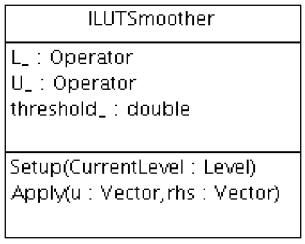

Fig. 9. Class diagram of a smoother object.

be computed for ILU smoothers. The smoother object stores data that depend on the level of the hierarchy. In general, the setup phase of a smoother is very limited, but this phase can be costly for some smoothers (like ILU). When the same smoother is used for both pre and post smoothing, the setup phase must be done only once per level.

Smoother objects of each level can be instantiated using again the factory design pattern: a smoother factory can generate smoother objects for each multigrid level and call the `Setup()` method of newly created smoothers to initialize them. Pre- and post-smoothers are generated at the same time to detect when the second setup phase can be avoided. In the following, we explain that for generating smoothers, we can in fact use a unique factory for every type of smoother. This is achieved by basing the factory design on the *prototype design pattern* [25].

If we used only the classic factory design pattern for smoothers, a different factory would have to be developed for each type of smoother. The main issue is that smoother objects have different types and different parameters (number of iterations, relaxation parameters, threshold of ILU, . . .). The prototype design pattern allows us to use only one generic smoother factory. In addition, users can manipulate directly smoother objects rather than factories: the generic factory does not appear in the interface and is only used internally.

In the prototype design pattern, factories use prototypes as models to make objects. The prototype gives the exact type of object to build (e.g., ChebySmoother, ILUSmoother). Smoother parameters are not held by factories. Instead, the parameters are stored directly in the prototypes. A prototype is like an unfinished object, as it never stores data relative to a level. To make a functional smoother, the prototype must be cloned, and the clone's `Setup()` method must be called. The main advantage of the prototype pattern is that the factories do not need to know anything about the object being created, i.e., factories can be generic.

Different smoother prototypes can be defined for each level, making it possible to use different parameters (or even different smoothers) on each level.

4. Leveraging TRILINOS capabilities

The TRILINOS framework provides a number of mature numeric and software development capabilities. Figure 10 illustrates the TRILINOS software stack and the relationship of MueLu to other libraries. MueLu has a required dependency on TEUCHOS and a linear algebra package, currently either EPETRA or TPETRA. Dependencies on other packages are only optional.

In Section 4.1, we summarize existing numerical capabilities in TRILINOS that MueLu is designed to leverage. In Section 4.2 we describe the thin layer that separates MueLu from the underlying linear algebra library.

4.1. Multigrid algorithm components from Trilinos

The main MueLu multigrid algorithm component that depends heavily on other Trilinos packages is the smoother used on each level. Smoothers can be iterative methods, such as Gauss–Seidel, incomplete fac-

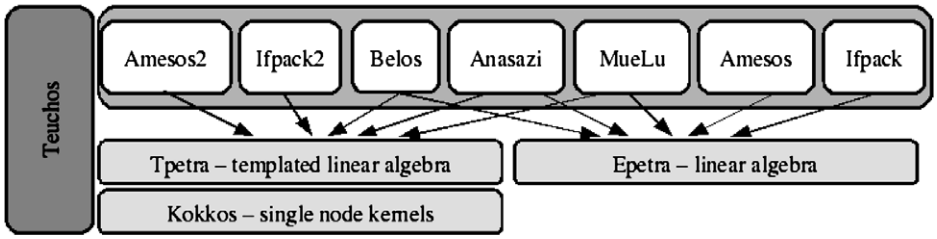

Fig. 10. Part of the TRILINOS software stack. Arrows denote which packages can use the linear algebra packages.

Table 1
TRILINOS packages implementing solvers that MueLu can use

Package	Algorithms	EPETRA	TPETRA
IFPACK	Preconditioners (Gauss–Seidel, Chebyshev, ILU, ...)	x	
IFPACK2	Preconditioners (Gauss–Seidel, Chebyshev, ILU, ...)		x
AMESOS	Sparse direct solvers (Serial and Parallel)	x	
AMESOS2	Sparse direct solvers (Serial and Parallel)	x	x
AZTECOO	Smoothers and Krylov methods	x	
BELOS	Krylov methods (CG, GMRES, ...)	x	x

torizations, Krylov methods, or even direct solvers. As TRILINOS already has efficient implementations of all these smoother types, MueLu is designed to use these libraries instead of having redundant internal implementations. Table 1 enumerates the smoothers in TRILINOS that MueLu can use.

MueLu leverages the ANASAZI package to provide efficient parallel methods for eigenvalue estimates [1, 2]. The needs for such estimates can arise during smoother setup (e.g., damped Jacobi, Chebyshev). If the prolongator construction is based on smoothed aggregation principles, then an accurate estimate of the largest eigenvalue is necessary in the damped Jacobi iteration that generates the interpolation matrix [26].

MueLu is designed to use ZOLTAN [5,7] during the multigrid setup phase to load balance coarse level operators based on nonzero metrics. In our experience, such load balancing is absolutely essential for good large scale parallel multigrid performance. MueLu is also designed to use ZOLTAN during smoothed aggregation prolongator construction to create sets of fine grid unknowns. These sets form the coarse grid unknowns. While these groupings are typically generated by the multigrid package itself, ZOLTAN is much more efficient at doing this when the groupings are desired to be much larger than in standard smoothed aggregation.

4.1.1. Parallel linear algebra

Input data to a sparse linear solver are mainly linear algebra objects like matrices and vectors. Two generations of a parallel linear algebra library coexist in TRILINOS: EPETRA and TPETRA. EPETRA is currently more heavily used, but we anticipate that new applications will write to the TPETRA interfaces, and that some existing applications will migrate to TPETRA. As a result, MueLu supports both EPETRA and TPETRA objects as input arguments to accommodate as many users as possible.

Both EPETRA and TPETRA support sparse linear algebra matrix operations that are required by linear solvers. However, EPETRA is limited to double scalar type and 32-bit integer ordinal type (for indexing). TPETRA is templated on both scalar and ordinal types. Additionally, TPETRA may use node-optimized kernels from the KOKKOS package. KOKKOS allows code, once written, to be run on any parallel node, regardless of architecture (CPU, GPU, ...). KOKKOS provides an abstract notion of compute node and a set of constructs for parallel computing operations. When run with TPETRA, MueLu benefits from any package (e.g., IFPACK2, AMESOS2) that uses KOKKOS to exploit the performance of new hardware architectures (multicore, GPU, ...). MueLu algorithms could also be adapted to new hardware architectures by using KOKKOS directly, but this would impact only specific internal algorithms and not the global design of MueLu.

4.2. The unified linear algebra layer

As explained in Section 4.1.1, MueLu can currently use either EPETRA or TPETRA as its underlying lin-

ear algebra package. However, a major design goal was to create a single extensible linear algebra interface, subject to several constraints. First, the layer should not significantly limit performance of the algorithms. Second, MueLu developers should be able to write algorithms completely in terms of this interface without knowledge of the actual underlying linear algebra package. Third, the layer should allow access to a linear algebra object in a fashion other than what its native storage (point, constant blocked, variable blocked) dictates. In the following sections, we discuss each of these constraints.

4.2.1. Lightweight, high performance interface

While C++ supports polymorphism at compile-time via the *Traits* technique [21], we decided to interface MueLu and the linear algebra packages by using runtime polymorphism through inheritance and interfaces. Our decision was based on several factors:

- Inheritance is straightforward for new users to understand. It simplifies the MueLu interface. Simple (virtual) interfaces are defined whenever possible, with default derived classes. More complicated interfaces are possible through inheritance.
- Inheritance supports extensibility of storage and functionality to objects of the linear algebra libraries. This capability is used on Level 2 of the MueLu linear algebra layer (discussed further in Section 4.2.2).
- The compile time overhead is much lower.
- Tests showed that there was little difference in performance between compile-time and run-time polymorphism for common use cases.

For efficiency reasons, we use internally the data format provided by users. For example, if the application matrix is stored as a TPETRA compressed row storage matrix (CRS), then we keep it in this format and create all coarse grid matrices as TPETRA CRS matrices. A thin abstraction layer provides an unified interface to the linear algebra packages for allowing MueLu developers to write code only once for both the EPETRA and TPETRA libraries. While support for both EPETRA and TPETRA already exists, a user working with a different data format simply needs to write an *adapter* allowing MueLu to work directly with the native format.

For the thin abstraction layer, we decided to create a new interface similar to that of TPETRA. This was based on the following considerations:

(1) Although some TRILINOS packages have existing interfaces for the internal use of linear algebra packages, most of these focus primarily on vectors and matrix–vector products. In multigrid, however, there are many linear algebra objects (maps, graphs, matrices, vectors, multivectors) that must be accessed in point or row fashion. Thus, the existing interfaces are not full-featured enough.
(2) The similarity of the abstract layer to that provided by TPETRA means that developers do not need to learn a completely new interface.
(3) As TPETRA and EPETRA have somewhat different native interfaces, the work for coding the abstraction layer is limited to writing an adapter from the EPETRA interface to the TPETRA interface.
(4) The TPETRA interface facilitates leveraging advanced C++ features such as templates and RCPs.

As this linear algebra layer enables the use of either TPETRA or EPETRA, we have named this layer XPETRA.

4.2.2. Matrix formats and the view mechanism

In addition to separating MueLu from the particular syntax of the underlying linear algebra constructs, the XPETRA linear algebra layer is designed to allow access to a matrix as if it were stored in some format other than its native storage format. For this purpose, we have developed a concept called a matrix *view*. A *view* of a matrix consists of the row access method, matrix–vector multiply, and maps associated with the matrix.

The *view* mechanism confers a number of benefits. It has been our experience that applications often store logically blocked matrices in a point matrix format. For example, a compressed row storage (CRS) format might be used for a PDE system with 5 degrees of freedom per node, instead of using a constant block storage format. For both algorithmic and performance reasons, it may still be desirable to treat the matrix as if it were blocked without making a deep copy of the data. Due to convergence concerns, it may be advantageous to use a block smoother, access the block diagonal, produce blocked grid transfers, or use some other block algorithm. Blocked formats can also lead to better performance in linear algebra kernels like the matrix–matrix multiply because of reduced indirect indexing. Access requirements may also change from one phase to the next, due to algorithmic reasons.

The *view* mechanism is embedded in the MueLu-`Operator` interface. The `Operator` class is an in-

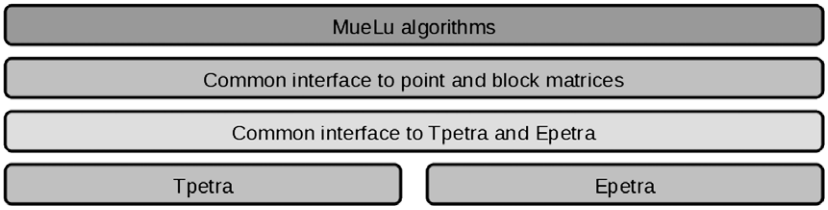

Fig. 11. Interfaces between MueLu and TRILINOS linear algebra libraries.

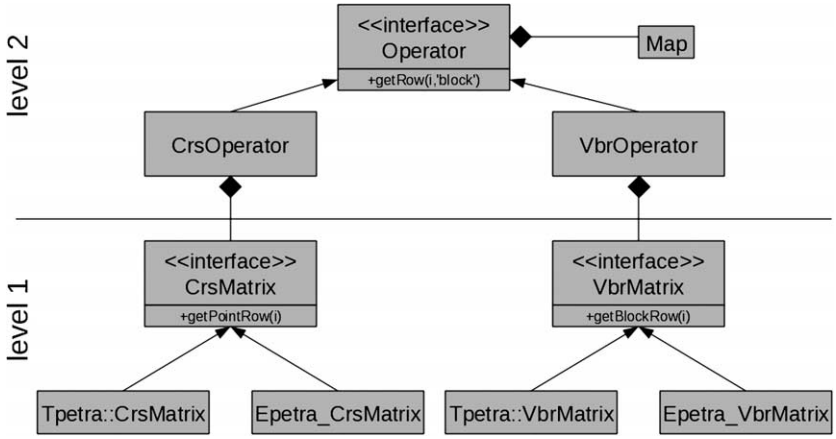

Fig. 12. Class diagram of the XPETRA linear algebra layer.

terface between the previously described abstraction layer for linear algebra libraries and MueLu algorithms. Figures 11 and 12 summarize the general design of the linear algebra layer. It finally consists of two layers: the low-level layer (layer 1) provides an abstraction of EPETRA and TPETRA, and the top-level layer (layer 2) provides an abstraction of the native data format.

The `Operator` interface and the *view* mechanism allow most algorithms to be written just once, yet support both point and block matrices. In particular, the `Operator` interface design supports:

- Viewing a CrsMatrix as a block matrix with a constant block size of 1 to apply block algorithms to point matrices.
- Viewing a CrsMatrix as a block matrix with a compatible blocking scheme, if user data are stored in CRS format but arise from a system of PDEs.
- Optionally viewing block matrices with a block scheme different but compatible to the native block format.

Using a *view* different than the native storage format may incur a significant performance penalty but may be mandatory to apply some algorithms. In compute intensive kernels, format-specific implementations may be necessary for good performance. However, the *view* can be very convenient in non-intensive kernels, especially in the multigrid setup phase.

View mechanism details. The default *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 *View* table which is just a list of *view* objects. Each *view* is fully described by two Map objects (for rows and columns).

It may also be that the multigrid method needs only a different *view* of the matrix diagonal, and not the entire matrix. This could happen, for example, in either the smoother or during the prolongator construction. For this reason, we also have a separate object, the `Diagonal`, to encapsulate just the matrix diagonal *view*. Figure 13 illustrates the use of the *view* mechanism. The basic idea is that `GetRow()` or `GetDiagonal()` has a different effect, depending on the *view*. Note that only the classes `Operator` and `Diagonal` are directly used in the other parts of MueLu. The concrete implementations of the `Operator` interface have to create the temporary block structure that is needed to convert data from the native format to a virtual *view*.

```
// Linear problem
RCP< Tpetra::CrsMatrix<double> > A = ...; // Scalar matrix

// MueLu operator. Default view is 'point' (1x1 block)
RCP< MueLu::Operator > Op = rcp (new CrsOperator(A));

// Create a 2x2 block representation of the matrix

Op.CreateConstBlkView("2x2", 2, 2);

// The 2x2 can be used everywhere by default by using:
// Op.SetDefaultView('2x2');

// Alternatively, it can be done on a case by case basis:

// Multigrid hierarchy
Hierarchy H(Op);

// The 2x2 block view is used during the aggregation and
    SaPFactory
aggregationFact = rcp(new UCAggregationFactory());

tentativePFact = rcp(new TentativePFactory(aggregationFact));
prolongationFact = rcp(new SaPFactory(tentativePFact));
prolongationFact->SetDiagonalView("2x2");
prolongationFact->SetView("2x2");

// Smoothers still use the 1x1 view
smootherProto = rcp(new Ifpack2Smoother<double>());
smootherFact = rcp(new SmootherFactory(smootherProto));
```

Fig. 13. Example of switching *views*.

Finally, we note that operations between two `Operators` (e.g., `OpA + OpB`) with differing internal data storage could be performed by using the point format as the common language between the two. This is likely to be slow, however. It might be more efficient to instead find the "greatest common block format" by comparing blocking schemes. Generally, MueLu stores matrices natively in the same format, thus avoiding this issue.

5. Application

In this section we describe how our framework can be easily extended to support multigrid approaches based on energy minimization. Such algorithms are very versatile and can be adapted to a variety of problems. The main implementation challenge is to preserve this intrinsic flexibility and to allow to reuse and reorganization of software components to help build methods for other problems.

As the focus of this discussion is not numerical results, but rather the ease with which such algorithms can be deployed and the resulting flexibility, we discuss the algorithm only at a high level. The energy minimization algorithm takes as input a desired prolongator sparsity pattern and user-supplied near-nullspace components. The algorithm generates a prolongator P that meets the sparsity criterion, interpolates the near-nullspace components (or a subset thereof), and whose columns P_i minimize the equation $\Sigma_i \|P_i\|_e$, where the norm is an *energy* norm defined by the Krylov minimization scheme. The details of the minimization process can be found in [22].

From a software design perspective, the energy-minimization process is fairly simple, and the power of the method resides in the freedom of choosing its input arguments. To reflect this freedom of choice, the prolongator factory implementing the energy-minimization algorithm relies on several auxiliary modules. The construction of prolongators can be completely controlled by specifying these auxiliary modules. Figure 14 shows the dependency graph of the factory involved in the construction of an energy minimization prolongator. Most of the modules are factories, as they

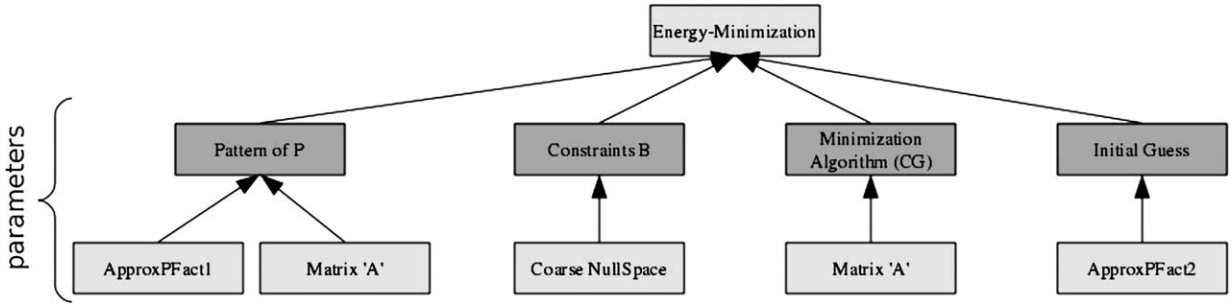

Fig. 14. Dependency graph showing factories involved in construction of an energy minimization prolongator.

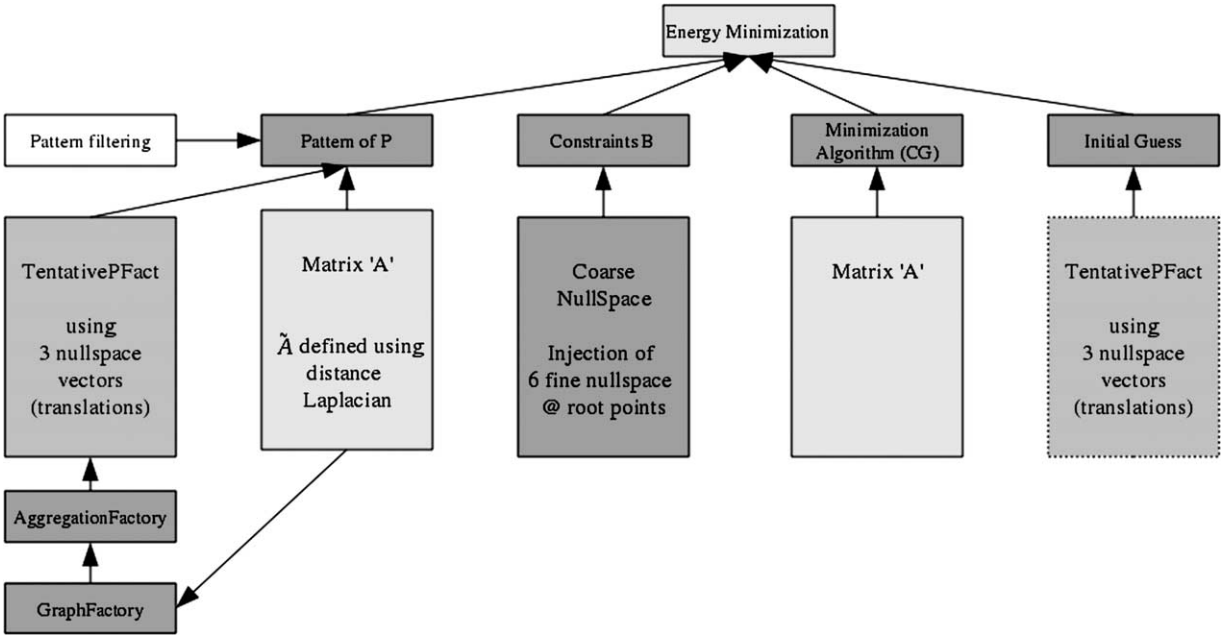

Fig. 15. Example of configuration of the energy-minimization algorithm to solve 3D elasticity problems.

represent algorithms producing inputs for the prolongator factory according to the multigrid level.

- Pattern factory – generates a sparsity pattern that the prolongator must satisfy.
- Constraint factory – generates a system of constraints that ensures certain given near nullspace vectors are well interpolated.
- Minimization algorithm – Krylov method that will guide the minimization process.
- Initial guess factory – generates an initial guess for the prolongator and the minimization process.

The highly modular design of the energy-minimization implementation allows experimentation with all of the parameters. In the following, we focus on the application of energy-minimization approach to solve a matrix system arising from linear elasticity. Figure 15 presents a possible setup configuration of the energy-minimization algorithm to solve 3D elasticity problems. Such problems have six near-nullspace components, and methods likes smoothed aggregation generate coarse grids with six degrees of freedom per nodes (even if the initial problem only has three degrees of freedom). The solver configuration presented here is inspired by smoothed aggregation, but the coarse grid size is reduced by using an energy-minimization algorithm. Our configuration reuses part of the smoothed aggregation implementation. In particular, the tentative prolongator factory generates both the sparsity pattern of P and an initial guess for the minimization process. The nullspace used by the tentative prolongator factory is restricted to three vectors (to limit the number of de-

grees of freedom per node on the coarse grid), but interpolation of all six near-nullspace components is enforced by the minimization algorithm.

This solver configuration leverages some of the most advanced features of our framework. It demonstrates the modularity of our framework and also justifies why we carefully defined the management of data flow between factories. Firstly, it heavily reuses part of existing multigrid methods and demonstrates the advantages of our modularity approach. The smoothed aggregation components (see Section 3.2.1) were not designed with this use case in mind but can handle it without any modification. Furthermore, even if a module appears several times on the calling tree of the method, that module is called only once because of the reference counter mechanism of the `Level` class. This is important here for the tentative prolongator factory that is used both for the sparsity pattern and the initial guess. Factories requesting outputs of the tentative prolongator factory do not have to know if these outputs will be reused later on. In this particular application, pattern and initial guess are generated from the same factory instance. In other methods, they may also come from different instances of the same factory (with different parameters or submodules). This means that they each will need similar inputs and create similar output data. In this case, data associated with one instance are kept distinct from the data associated with the other.

One should also notice that the tentative prolongator factory produces coarse nullspace vectors but that another definition of the nullspace is created by a different factory for the energy-minimization algorithm. Again, as output of factories are distinguishable, modules can select different nullspace information. The same mechanism is also in action to distinguish the tentative prolongator and the final prolongator. Finally, an aggregation algorithm is required by the tentative prolongator factory, but as each module is designed to be self-sufficient, a default algorithm will be called automatically if none is provided. Hence, users still have full control over aggregation parameters if need be.

6. Conclusion

In this paper we have presented a flexible design for a multigrid preconditioning library, MueLu. The factory pattern is used throughout the design, which supports both current core solver methods as well as future algorithm research. The design is general enough that both algebraic and geometric algorithms are possible. The library interfaces support a wide range of users. The multigrid preconditioners can be customized by simple interfaces that use parameter lists. More advanced interactions are supported by directly creating and modifying factories for each of the multigrid algorithm components. MueLu leverages many existing mature Trilinos libraries, due in large part to a lightweight linear algebra layer called XPETRA that separates MueLu from the underlying linear algebra library.

References

[1] 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. Soft.* **36** (2009), 13:1–13:23.

[2] C.G. Baker, U.L. Hetmaniuk, R.B. Lehoucq and H.K. Thornquist, Anasazi block eigensolver package documentation, available at: http://trilinos.sandia.gov/trilinos/packages/anasazi/.

[3] R. Bartlett, Teuchos::RCP beginner's guide (an introduction to the Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++), Technical Report 2004-3268, Sandia National Laboratories, 2007, available at: http://trilinos.sandia.gov/RefCountPtrBeginnersGuideSAND.pdf.

[4] R. Bartlett, Teuchos C++ memory management classes, idioms, and related topics: the complete reference, Technical Report 2010-2234, Sandia National Laboratories, 2010, available at: http://www.cs.sandia.gov/~rabartl/TeuchosMemoryManagementSAND.pdf.

[5] E. Boman, K. Devine, L.A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag and W. Mitchell, Zoltan home page, 1999, available at: http://www.cs.sandia.gov/Zoltan.

[6] W.L. Briggs, V.E. Henson and S.F. McCormick, *A Multigrid Tutorial*, 2nd edn, Soc. Ind. Appl. Math., Philadelphia, PA, 2000.

[7] K. Devine, E. Boman, R. Heaphy, B. Hendrickson and C. Vaughan, Zoltan data management services for parallel dynamic applications, *Comput. Sci. Eng.* **4** (2002), 90–97.

[8] R.D. Falgout, J.E. Jones and U.M. Yang, The design and implementation of hypre, a library of parallel high performance preconditioners, in: *Numerical Solution of Partial Differential Equations on Parallel Computers*, A.M. Bruaset and A. Tveito, eds, Springer, New York, 2006, pp. 267–294.

[9] R.D. Falgout and U.M. Yang, Hypre: a library of high performance preconditioners, in: *International Conference on Computational Science*, Lecture Notes in Computer Science, Vol. 2331, Springer, Berlin, 2002, pp. 632–641.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Vol. 206, Addison-Wesley, Reading, MA, 1995.

[11] M. Gee, C. Siefert, J. Hu, R. Tuminaro and M. Sala, ML 5.0 smoothed aggregation user's guide, Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[12] V.E. Henson and U.M. Yang, BoomerAMG: a parallel algebraic multigrid solver and preconditioner, *Appl. Num. Math.* **41** (2000), 155–177.

[13] M. Heroux, C. Baker and A. Williams, Tpetra: next-generation templated Petra, available at: http://trilinos.sandia.gov/trilinos/packages/tpetra.

[14] M. Heroux, R. Bartlett, V.H.R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring and A. Williams, An overview of Trilinos, Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[15] M. Heroux, R. Hoekstra, P. Sexton, B. Spotz, J. Willenbring and A. Williams, Epetra: linear algebra services package, 1999, available at: http://trilinos.sandia.gov/trilinos/packages/epetra/.

[16] M.A. Heroux, C.G. Baker, R.A. Bartlett, K. Kampschoff, K.R. Long, P.M. Sexton and H.K. Thornquist, Teuchos: the Trilinos tools library, available at: http://trilinos.sandia.gov/trilinos/packages/teuchos/.

[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. Soft.* **31** (2005), 397–423.

[18] M.A. Heroux and J.M. Willenbring, Trilinos users guide, Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[19] M.A. Heroux, J.M. Willenbring and R. Heaphy, Trilinos developers guide, Technical Report SAND2003-1898, Sandia National Laboratories, 2003.

[20] M.A. Heroux, J.M. Willenbring and R. Heaphy, Trilinos developers guide part II: ASCI software quality engineering practices version 1.0, Technical Report SAND2003-1899, Sandia National Laboratories, 2003.

[21] N. Myers, *A New and Useful Template Technique: "Traits"*, SIGS Publications, New York, NY, 1996.

[22] L. Olson, J. Schroder and R. Tuminaro, A general interpolation strategy for algebraic multigrid using energy minimization, *SIAM J. Sci. Comput.* **33** (2011), 966–991.

[23] M. Sala, An object-oriented framework for the development of scalable parallel multilevel preconditioners, *ACM Trans. Math. Soft.* **32** (2006), 396–416.

[24] M. Sala, M.A. Heroux and D.M. Day, Trilinos tutorial, Technical Report SAND2004-2189, Sandia National Laboratories, 2004.

[25] A. Shalloway and J.R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, 1st edn, Addison-Wesley, Reading, MA, 2001.

[26] P. Vaněk, J. Mandel and M. Brezina, Algebraic multigrid-based on smoothed aggregation for second and fourth-order problems, *Computing* **56** (1996), 179–196.

[27] J. Wan, T. Chan and B. Smith, An energy-minimizing interpolation for robust multigrid methods, *SIAM J. Sci. Comput.* **21** (2000), 1632–1649.