MueLu – A Flexible, Parallel Multigrid Framework

Trilinos User Group Meeting
Oct. 30 – Nov. 1, 2012

Jonathan Hu

2012-9327P
Development Team

Jonathan Hu
Jeremie Gaidamour
Andrey Prokopenko
Tobias Wiesner (TU Munich)
Ray Tuminaro
Chris Siefert
Outline

• Design and Motivation
• User interfaces
• Case study: smoothed aggregation
  – Reuse possibilities
Design and Motivation
Motivation for a New Multigrid Library

• **Trilinos already has mature multigrid library, ML**
  – Algorithms for Poisson, Elasticity, Petrov-Galerkin, H(curl), H(div)
  – Algorithms have been exercised extensively.
  – Broad user base

• **However …**
  – ML weakly linked to other Trilinos capabilities (e.g., smoothers)
  – C-based, only scalar type “double” supported explicitly
  – Over 50K lines of source code
    • Maintainability, extensibility
Objectives for New Multigrid Framework

• **Templating** on scalar, ordinal types

• **Advanced architectures**
  – Kokkos support for various compute node types
    • Hybrid parallelism: MPI, MPI+threads, MPI+MPI
    • GPUs eventually

• **Extensibility**
  – Facilitate development of other algorithms
    • Energy minimization methods
    • Geometric, classic algebraic multigrid, …
  – Ability to combine several types of multigrid

• **Preconditioner reuse**
  – Reduce setup expense
Multigrid Basics

• Two main components
  – Smoothers
    • Approximate solves on each level
    • “Cheaply” reduces particular error components
    • On coarsest level, smoother = $A_i^{-1}$ (usually)
  – Grid Transfers
    • Moves data between levels
    • Must represent components that smoothers can’t reduce

• Algebraic Multigrid (AMG)
  – AMG generates grid transfers
  – AMG generates coarse grid $A_i$’s

\[ Au = f \]
\[ \Omega[0] \]
\[ \Omega[1] \]
\[ \Omega[2] \]
\[ \Omega[\text{maxlevel}] \]
Current MueLu Capabilities

• Grid Transfer Algorithms
  – Smoothed aggregation, Petrov Galerkin

• Smoothers
  – SOR, ILU, Polynomial (Ifpack, Ifpack2)

• Direct solvers
  – KLU, SuperLU, SuperLUDist (Amesos, Amesos2)

• Sparse linear algebra (Epetra, Tpetra)

• Krylov acceleration (Belos, AztecOO)
• Wrapper for Epetra and Tpetra
  – Based on Tpetra interfaces
  – Allows unified access to either linear algebra library

• Layer concept:
  – **Layer 2**: blocked operators
  – **Layer 1**: operator views
  – **Layer 0**: low level E/Tpetra wrappers (automatically generated code)

• MueLu algorithms are written using Xpetra
Design Overview

• MueLu makes heavy use of “factory” pattern
  – Factories: classes that generate objects

• Preconditioner is created by chaining together factories that create grid transfers, smoothers, coarse grid Galerkin triple-matrix product

• FactoryManager manages these dependencies

• User is not required to specify these dependencies (or even know they exist).
User Interfaces
MueLu – User Interfaces

• **MueLu can be customized as follows:**
  – XML input files
  – Parameter lists (key-value pairs)
  – Directly through C++ interfaces

• **New/casual users**
  – Minimal interface
  – Sensible defaults provided automatically

• **Advanced users**
  – Can customize or replace any component of multigrid algorithm.
MueLu – A Simple C++ Example

// Creation of fine matrix A, solution X, right-hand side B not shown

// Allocate hierarchy object and insert A
Hierarchy H(fineA);
H.Setup();
H.Iterate(B,nits,X);

• Generates smoothed aggregation multigrid preconditioner.
• Uses reasonable defaults.
• As we’ll see, these can changed easily.
Customizing the Preconditioner

// Creation of fine matrix A, solution X, right-hand side B not shown

// Allocate hierarchy object and insert A
Hierarch y H(fineA);

RCP<TentativePFactory> ProlongatorFact = rcp( new TentativePFactory() );
Teuchos::ParameterList smootherParamList;
smootherParamList.set("Chebyshev: degree", 3);
RCP<SmootherPrototype> smootherPrototype = rcp( new TrilinosSmoother("Chebyshev", smootherParamList) );

FactoryManager M;
M.SetFactory("P",ProlongatorFact);
M.Set("Smoother",Smoothe rPrototype);

H.Setup(M);

int its=10;
H.Iterate(B,nits,X);
Customizing the Preconditioner

// Creation of fine matrix A, solution X, right-hand side B not shown

// Allocate hierarchy object and insert A
Hierarchy H(fineA);

RCP<TentativePFactory> ProlongatorFact = rcp( new TentativePFactory() );
Teuchos::ParameterList smootherParamList;
smootherParamList.set("Chebyshev: degree", 3);
RCP<SmootherPrototype> smootherPrototype = rcp( new TrilinosSmoother("Chebyshev", smootherParamList) );

FactoryManager M;
M.SetFactory("P",ProlongatorFact);
M.Set("Smoother",SmoootherPrototype);

H.Setup(M);

int its=10;
H.Iterate(B,nits,X);

• Use unsmoothed prolongator
  – Rcp == smart pointer
Customizing the Preconditioner

// Creation of fine matrix A, solution X, right-hand side B not shown

// Allocate hierarchy object and insert A
Hierarchy H(fineA);

RCP<TentativePFactory> ProlongatorFact = rcp(new TentativePFactory());

Teuchos::ParameterList smootherParamList;
smootherParamList.set("Chebyshev: degree", 3);
RCP<SmootherPrototype> smootherPrototype = rcp(new TrilinosSmoother("Chebyshev", smootherParamList));

FactoryManager M;
M.SetFactory("P", ProlongatorFact);
M.Set("Smoother", SmootherPrototype);

H.Setup(M);

int its=10;
H.Iterate(B,nits,X);

• Use degree 3 polynomial smoother
  – Parameter list == key/value pairs
  – Smoother prototype
Customizing the Preconditioner

// Creation of fine matrix A, solution X, right-hand side B not shown

// Allocate hierarchy object and insert A
Hierarchy H(fineA);

RCP<TentativePFactory> ProlongatorFact = rcp( new TentativePFactory() );
Teuchos::ParameterList smootherParamList;
smootherParamList.set("Chebyshev: degree", 3);
RCP<SmootherPrototype> smootherPrototype = rcp( new TrilinosSmoother("Chebyshev", smootherParamList) );

FactoryManager M;
M.SetFactory("P",ProlongatorFact);
M.Set("Smoother",SmoothenPrototype);

H.Setup(M);

int its=10;
H.Iterate(B,nits,X);

• Register changes with Factory Manager and pass to Setup.
The Factory Manager

- Holds default factories to be used during multigrid setup.
- Can have one FactoryManager per level.
- User can selectively specify alternatives.
  
  ```c
  FactoryManager M;
  M.SetFactory("Aggregation", UCAggFact);
  ```
- The hierarchy set up process queries the FactoryManager for proper factory for each algorithmic component.
Accessing MueLu Through XML

```cpp
//read in XML file...

ParameterListInterpreter mueLuFactory(xmlFileName);
RCP<Hierarchy> H = mueLuFactory.CreateHierarchy();
H->GetLevel(0)->Set("A", A);

mueLuFactory.SetupHierarchy(*H);

int nIts = 10;
H->Iterate(*B, nIts, *X);
```
Case Study: Smoothed Aggregation
Multigrid
Smoothed Aggregation Setup

- Group fine unknowns into aggregates to form coarse unknowns
Group fine unknowns into *aggregates* to form coarse unknowns

Partition given nullspace $B_{(h)}$ across aggregates to have local support
Smoothed Aggregation Setup

- Group fine unknowns into aggregates to form coarse unknowns

- Partition given nullspace $B_{(h)}$ across aggregates to have local support

- Calculate $QR = B_{(h)}$ to get initial prolongator $P^{tent} (= Q)$ and coarse nullspace ($R$).

- Form final prolongator $P^{sm} = (I - \omega D^{-1}A)P^{tent}$
Case Study: Smoothed Aggregation

- Possible call sequences to generate $p^{sm}$

1) $\text{PFact} = \text{SaPFactory}()$

2) $\text{PtentFact} = \text{TentativePFFactory}()$
   $\text{PFact} = \text{SaPFactory}(\text{PtentFact})$

3) $\text{AggFact} = \text{AggregationFactory}()$
   $\text{Ptent} = \text{TentativePF}(\text{AggFact})$
   $\text{PFact} = \text{SaPFactory}(\text{Ptent})$

- Data dependencies must be maintained between factories.
Management of Data Dependencies

- Level class manages data storage
- Factories exchange data by taking Level classes as arguments to Build method:
  - Build(currentLevel) or
  - Build(fineLevel,coarseLevel)
- Factories declare on Level the data that they require, along with generating factories, or FactoryManager provides generating strategy.
Advantages of Data Management on Level

- Level manages data deallocation once all requests satisfied
- Generating factory does not need to know what other factories require data
- **Data reuse**
  - Any data (aggregates, $P$, ...) can be retained by user request for reuse in later runs.
  - Data can be retained for later analysis.
  - Almost any reuse granularity is possible.
AggFact = AggregationFactory();
Ptent = TentativePFactory(AggFact);
Pfact = SaPFactory(Ptent);

- Pfact registers with Level its need for $Ptent$, along with generating factory Ptent.
- Ptent registers with Level its need for aggregate data, along with generating factory (AggFact).
- AggFact generates aggregates, stores on Level.
- After Ptent accesses aggregates, Level frees data.
- After Pfact access $Ptent$, Level frees data.

User does not need to manage data dependencies.
Summary

• Current status
  – Copyrighted with open-source BSD style license
  – Part of publicly available Trilinos anonymous clone
  – We still support ML.

• Ongoing/Future work
  – New team member Andrey Prokopenko
  – Grid transfers based on constrained minimization (aka energy minimization)
  – Improving documentation, application interfaces
  – Big driver for FY13 is templated stack milestone requirements
  – Performance optimizations