SAND2011-2070C

Designing an extensible framework for multigrid algorithm research

Jeremie Gaidamour
Jonathan Hu, Chris Siefert, Ray Tuminaro

15th Copper Mountain Conference on Multigrid Methods
March 2011

Sandia National Laboratories

Outline

- Introduction / Motivation
- Software
  - MueMat
  - MueLu
- Design overview of the framework
  - Interface, simple usage example
  - Design philosophy (Smoothed-Aggregation)
  - Data structures
- How to exploit algorithmic flexibility of energy minimization methods
  - Energy-minimization algorithms
  - Implementation details
- Examples of advanced usages
- Summary

Motivation for a new library

Why a new MG package?

Using MG solver as a black-box is not enough for challenging problems:

- Systems of PDEs (filtering, complexity)
- Sharp/Thin material interfaces, Anisotropic phenomena
- Physics-based methods

⇨ We need an highly flexible framework to experiment with MG methods:

- Software flexibility:
  - Implementation of problem-specific improvement directly on the heart of the solver should be easy
  - Code reutilisability, adaptivity to other problems

- Algorithm flexibility:
  - A better control over all the solver parameters is needed

Motivation for a new library

Introducing algorithmic novelty:

- Central theme: **constraint-based energy-minimization**
  - More flexibility on the choice of inter-grid operators by using energy-minimization algorithms
  - Explicit control over sparsity patterns of grid transfer operators (complexity)

- Support block matrices, block diagonal and block algorithms (both constant and variable sizes) for multiphysics PDEs

Extensibility, flexibility for future research:

- Intentional design allows for variety of methods: geometric/classic/aggregation based methods and hybrid methods that chain together two or more different methods

- Exascale simulation: new R&D challenges
  - Challenges for scalable algorithms, even MG methods
  - Exploit multicore architectures, hybrid and heterogeneous machines...

  => Require dramatic changes on both software design and algorithms

Software

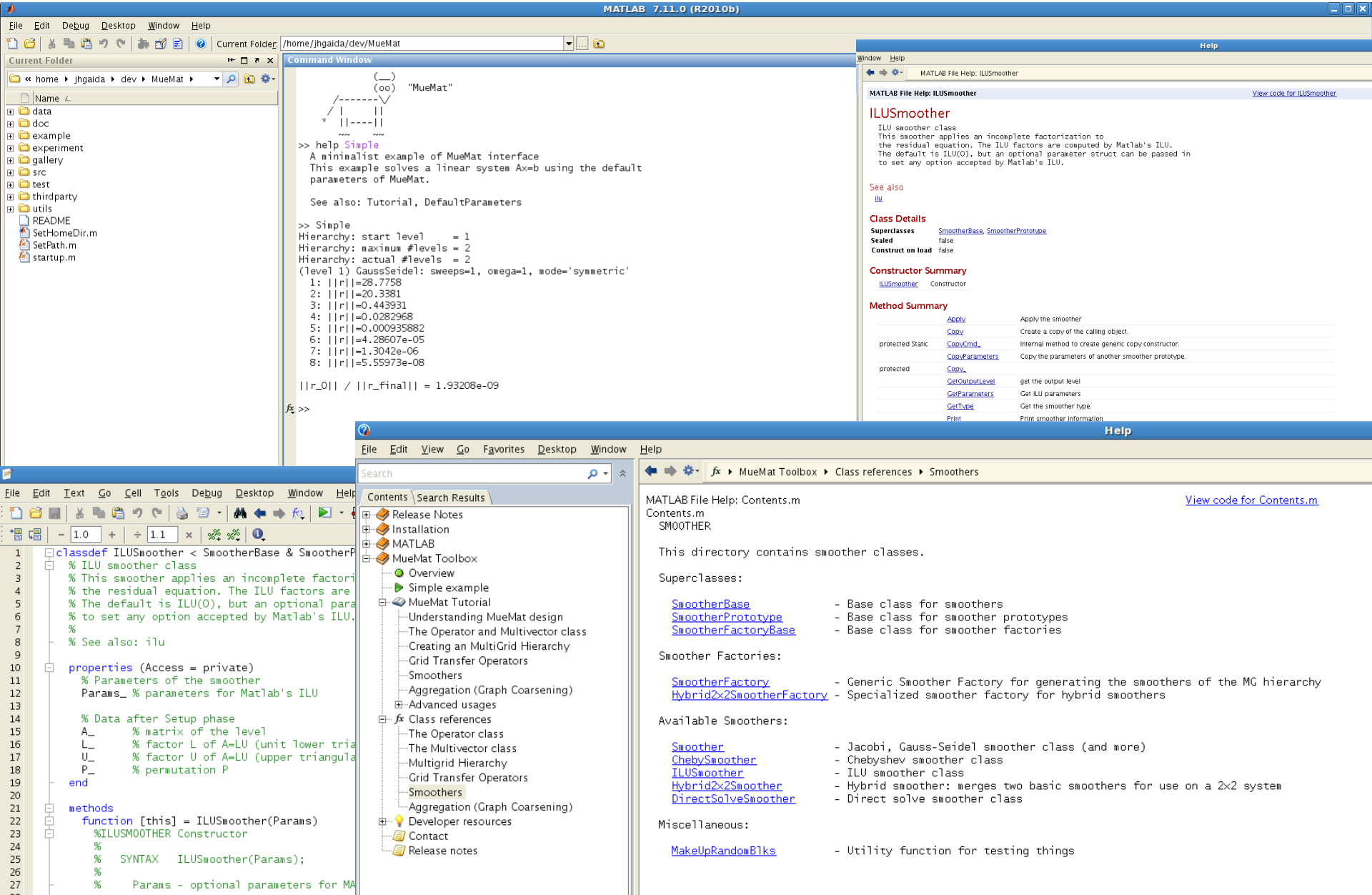
Software

- Two MG libraries:
  - MueMat: MATLAB version
  - MueLu: C++ (new Trilinos package)
- Simple interfaces (facades) provided for new users.
- Modularity for advanced users who want to reuse existing components and to tune MG methods for specific problems.
- Share the same *oriented object* design:
  - Easy control over all the component of the algorithm (aggregation, sparsity pattern, constraints, initial guess...) in the high level interface
  - Modular design allow swapping of algorithmic parts
- Other key features:
  - Support block matrices, block diagonal and block algorithms (both constant and variable sizes) for multiphysics PDEs
  - Intentional design allows for variety of methods: geometric/classic/aggregation based methods and hybrid methods that chain together two or more different methods

MueMat

- A sequential MATLAB version
- Why?
  - for rapid prototyping of the C++ version of the solver
  - for algorithm research and result analysis
  - for research collaboration, new staff, students...
- Turnkey capabilities: Smoothed Aggregation, Petrov-Galerkin SA (unsymmetric), Energy Minimization algorithm...
- User and developer documentation
- Will be released soon.

MueMat

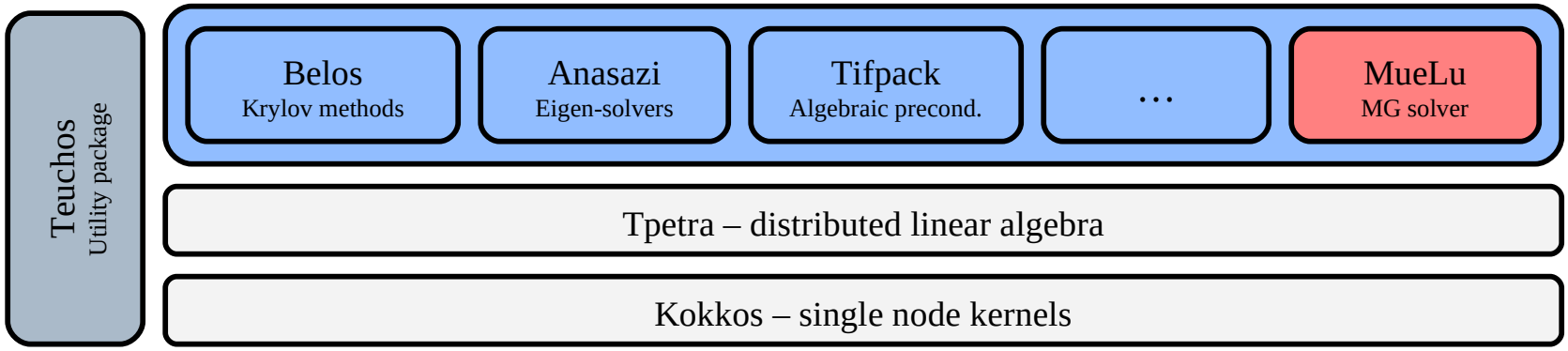

MATLAB 7.11.0 (R2010b)
File Edit Debug Desktop Window Help
Current Folder: /home/jhgaida/dev/MueMat
Current Folder
home jhgaida dev MueMat
Name
data
doc
example
experiment
gallery
src
test
thirdparty
utils
README
SetHomeDir.m
SetPath.m
startup.m
Command Window
(__)
(oo) "MueMat"
>> help Simple
A minimalist example of MueMat interface
This example solves a linear system Ax=b using the default
parameters of MueMat.
See also: Tutorial, DefaultParameters
>> Simple
Hierarchy: start level = 1
Hierarchy: maximum #levels = 2
Hierarchy: actual #levels = 2
(level 1) GaussSeidel: sweeps=1, omega=1, mode='symmetric'
1: ||r||=28.7758
2: ||r||=20.3381
3: ||r||=0.443931
4: ||r||=0.0282968
5: ||r||=0.000935882
6: ||r||=4.28607e-05
7: ||r||=1.3042e-06
8: ||r||=5.55973e-08
||r_0|| / ||r_final|| = 1.93208e-09
>>
Help
Window Help
MATLAB File Help: ILUSmoother
MATLAB File Help: ILUSmoother
View code for ILUSmoother
ILUSmoother
ILU smoother class
This smoother applies an incomplete factorization to
the residual equation. The ILU factors are computed by Matlab's ILU.
The default is ILU(0), but an optional parameter struct can be passed in
to set any option accepted by Matlab's ILU.
See also
ilu
Class Details
Superclasses SmootherBase, SmootherPrototype
Sealed false
Construct on load false
Constructor Summary
ILUSmoother Constructor
Method Summary
Apply Apply the smoother
Copy Create a copy of the calling object.
protected Static CopyCmd_ Internal method to create generic copy constructor.
CopyParameters Copy the parameters of another smoother prototype.
protected Copy_
GetOutputLevel get the output level
GetParameters Get ILU parameters
GetType Get the smoother type.
Print Print smoother information
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
classdef ILUSmoother < SmootherBase & SmootherP
% ILU smoother class
% This smoother applies an incomplete factori
% the residual equation. The ILU factors are
% The default is ILU(0), but an optional para
% to set any option accepted by Matlab's ILU.
%
% See also: ilu
properties (Access = private)
% Parameters of the smoother
Params_ % parameters for Matlab's ILU
% Data after Setup phase
A_ % matrix of the level
L_ % factor L of A=LU (unit lower tria
U_ % factor U of A=LU (upper triangula
P_ % permutation P
end
methods
function [this] = ILUSmoother(Params)
%ILUSMOOTHER Constructor
%
% SYNTAX ILUSmoother(Params);
%
% Params - optional parameters for MA
Help
File Edit View Go Favorites Desktop Window Help
Search
Contents Search Results
Release Notes
Installation
MATLAB
MueMat Toolbox
Overview
Simple example
MueMat Tutorial
Understanding MueMat design
The Operator and Multivector class
Creating an MultiGrid Hierarchy
Grid Transfer Operators
Smoothers
Aggregation (Graph Coarsening)
Advanced usages
Class references
The Operator class
The Multivector class
Multigrid Hierarchy
Grid Transfer Operators
Smoothers
Aggregation (Graph Coarsening)
Developer resources
Contact
Release notes
MueMat Toolbox Class references Smoothers
MATLAB File Help: Contents.m
View code for Contents.m
Contents.m
SMOOTHER
This directory contains smoother classes.
Superclasses:
SmootherBase - Base class for smoothers
SmootherPrototype - Base class for smoother prototypes
SmootherFactoryBase - Base class for smoother factories
Smoother Factories:
SmootherFactory - Generic Smoother Factory for generating the smoothers of the MG hierarchy
Hybrid2x2SmootherFactory - Specialized smoother factory for hybrid smoothers
Available Smoothers:
Smoother - Jacobi, Gauss-Seidel smoother class (and more)
ChebySmoother - Chebyshev smoother class
ILUSmoother - ILU smoother class
Hybrid2x2Smoother - Hybrid smoother: merges two basic smoothers for use on a 2x2 system
DirectSolveSmoother - Direct solve smoother class
Miscellaneous:
MakeUpRandomBlks - Utility function for testing things

MueLu

- Future package of the Trilinos project (to replace ML)
  - C++ - Object-oriented design
  - Massively parallel
  - Multicore and GPU aware
  - Templated types for mixed precision calculation (32-bit – 64-bit) and type complex

- Objective is to solve problem with billions of DOF on 100Ks of cores...

- Leverage the Trilinos software stack:



- Currently in development...

MueLu

- Supported native data storage:
  - Wrapper around Epetra or Tpetra matrices
  - Point and block matrices (CRS / VBR)

- Algorithms in MueLu written only once (for several matrix types)

- Data access:
  - independent from native storage format
    ie: block smothers can be applied to point matrices
  - view mechanism: any matrix can be accessed as a block or point matrix
  - point/block diagonal extraction

Overview of the framework

Overview of the framework

Basics:

- Object oriented design: *Prolongator (P), Restrictor (R), grid matrices (A, Ac) and smoothers are objects.*

- Extensive use of the factory method pattern:

  factories == classes whose main purpose is creation of objects.

- Algorithm of the setup phase:

```
For each level, do:
        P = ProlongatorFact.Build(...)
        R = RestrictionFact.Build(...)
        Ac = ProjectionFact.Build(...)
        PreSmoother = PreSmootherFact.Build(...)
        PostSmooter = PostSmootherFact.Build(...)
```

Basically, for each level of the MG hierarchy, user specifies how to build the level information by defining a set of factories. These factories work together to build P, R, A_{coarse}, ... during the setup phase.

Basic usage

User point of view:

- instantiate the factories that must be used (= define the MG method)
- levels are automatically built using the user provided factories
- data of levels are stored in a `Hierarchy` object (= list of levels)

Example:

Setup

```
PFact = SaPFactory();                    % Smoothed-Aggregation
RFact = TransposeFactory();              % R = P^T
SmootherFact = GaussSeidelFactory(); % Smoother = GaussSeidel
```

```
MgHierarchy = Hierarchy(A);
MgHierarchy.Populate(PFact, RFact, SmootherFact, 1, 5);
```

Solve

```
sol = MgHierarchy.Iterate(rhs, sol, 15);
```

Basic usage

User point of view:

- instantiate the factories that must be used (= define the MG method)
- levels are automatically built using the user provided factories
- data of levels are stored in a `Hierarchy` object (= list of levels)

Example:

Setup

```
PFact = SaPFactory();                    % Smoothed-Aggregation
RFact = TransposeFactory();              % R = P^T
SmootherFact = GaussSeidelFactory();     % Smoother = GaussSeidel
SmootherFact.SetNumIts(2);
```

Default behavior of factories easy to change

```
MgHierarchy = Hierarchy(A);
MgHierarchy.Populate(PFact, RFact, SmootherFact, 1, 5);
```

Solve

```
sol = MgHierarchy.Iterate(rhs, sol, 15);
```

Overview of the framework

The MG method is fully described by the hierarchy object and defining more complex MG methods is also possible:

- Ex 1: use different smoothers or coarsening according to the level

Setup

```
PFact = SaPFactory();
RFact = TransposeFactory();
GSSmootherFact = GaussSeidelFactory(2);
ILUSmootherFact = ILUFactory();
```

```
MgHierarchy = Hierarchy(A);
MgHierarchy.FillHierarchy(PFact, RFact,   1,3);
MgHierarchy.SetSmoothers(GSSmootherFact,  1,1);
MgHierarchy.SetSmoothers(ILUSmootherFact, 2,1);
```

Overview of the framework

The MG method is fully described by the hierarchy object and defining more complex MG methods is also possible:

- Ex 1: use different smoothers or coarsening according to the level
- Ex 2: hybrid methods mixing geometric/algebraic MG

Setup

```
P1Fact = GeoPFactory();
P2Fact = SaPFactory();
RFact  = TransposeFactory();
SmootherFact = GaussSeidelFactory();
```

```
MgHierarchy = Hierarchy(A);
MgHierarchy.FillHierarchy(P1Fact, RFact, 1,1);
MgHierarchy.FillHierarchy(P2Fact, RFact, 2,1);
MgHierarchy.SetSmoothers(SmootherFact);
```

Overview of the framework

The MG method is fully described by the hierarchy object and defining more complex MG methods is also possible:

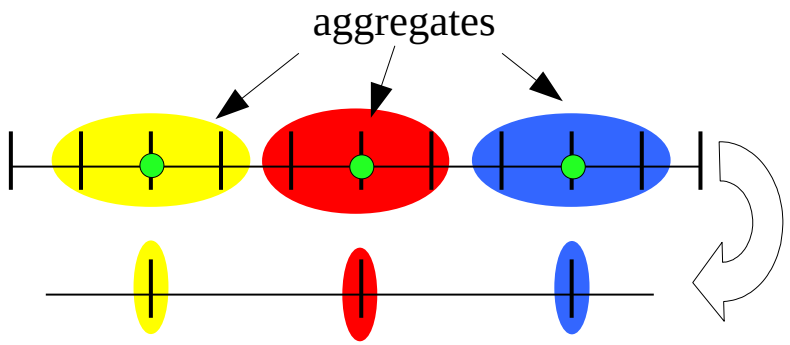
- Ex 1: use different smoothers or coarsening according to the level
- Ex 2: hybrid methods mixing geometric/algebraic MG

- Advantages:
  - no need of complex lists of parameters
  - object swapping easy to do (ex: changing the smoother)
  - adding new algorithm (like a new smoother) is straightforward. No need to understand/modify any existing code. Just create a new class and use it!

- Drawbacks:
  - users must understand the basics of a multigrid solver...
    ... but we also provide “facade” for common MG approaches
  - not that easy in reality ...
    ... specific requirements of some multigrid methods?
    ... what are the exact input/output of the Build() methods of factories?

Smoothed Aggregation details

- Coarsening:
  - Choose *root* points ●
  - Group fine unknowns into aggregates to form coarse unknowns



- Build initial guess P_0 that interpolates nullspace B:
  1. Partition nullspace into locally supported function
  2. Orthogonalized with QR, i.e., initial guess P_0 satisfies constraints

$$B = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ \vdots \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & & \\ 1 & & \\ 1 & & \cdots \\ & 1 & \\ & 1 & \\ & 1 & \\ & & \ddots \end{bmatrix}$$

- Improve P_0 with one step of damped Jacobi: $P = (I - \omega D^{-1} A) P_0$

Smoothed-Aggregation

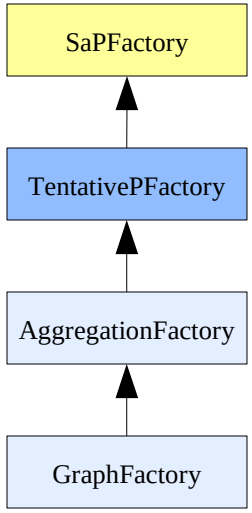
- Steps for building P:

```
Graph = CoalesceFact.Build(A);
Aggregates = AggregationFact.Build(Graph);
Ptent = TentativePFact.Build(Aggregates, NullSpace);
P = SaPFactory.Build(Ptent);
```

- Design decision:

```
SaPFactory.Build() {
    Ptent = this.TentativePFact_.Build();
    P = (I - wD^-1A) Ptent
}

TentativePFactory.Build() {
    Aggregates = this.AggregationFact_.Build();
    Ptent = ...
}
...
```



Sandia National Laboratories

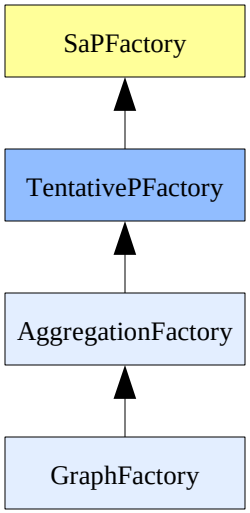
Design discussion

Usage:

```
1.PFact = SaPFactory();

2.PtentFact = TentativePFactory();
  PFact = SaPFactory(TentativePFactory);

3.AggregationFact = AggregationFactory();
  AggregationFact.SetTargetSize(10);
  PFact = SaPFactory(TentativePFactory(AggregationFact));
```



Advantages:

- No need for a driver to call factories in the right order
- Each factory can be use standalone and hide all the details because everything is embedded

Drawbacks:

- Complex (changing parameters)?
- What to do if data or factories are used several time?

Data structures

- In `Hierarchy`, each level is represented by an associative array `Level`
- Factories Build() methods share a common definition:

```
Build(FineLevel, CoarseLevel) or Build(CurrentLevel)
```

- Input/Output of all factories stored in a `Level` object:

```
TentativePFactory.Build(FineLevel, CoarseLevel) {
        NullSpace = FineLevel('NullSpace');
        ...
        ...
        CoarseLevel('P') = P;
        CoarseLevel('NullSpace') = CoarseNullSpace;
}
```

- It allows great flexibility on what factories can do:
  - data input:
    - Ex: Smoothed-Aggregation prolongation factories need NullSpace information, Geometric MG need geometry information...
  - data output:
    - Ex: Smoothed-Aggregation TentativePFactory compute the coarse nullspace that will be used to build the prolongator of the next level

Data structures

- Level also used as a “scratch pad” to store temporary data of the setup step: Example of temporary data: *Graph, Aggregates, P tentative, Coarse nullspaces, Auxiliary matrices*

- Data deallocation:
  - Deallocation of data in `Level` as soon as possible
  - Idea: Reference count pointers
    - Before the setup phase, factories declares 'input' data Example: `SaPFactory` need the prolongator produced by `TentativePFactory => counter++`
    - After the execution of `SaPFactory: counter--`
    - if `counter==0` free temporary data.
  - Data can be kept for plotting, debugging...

- When you write a factory, no need to know:
  - How your factory will be linked with other
  - If you have to allocate data you are using or deallocate data you are producing

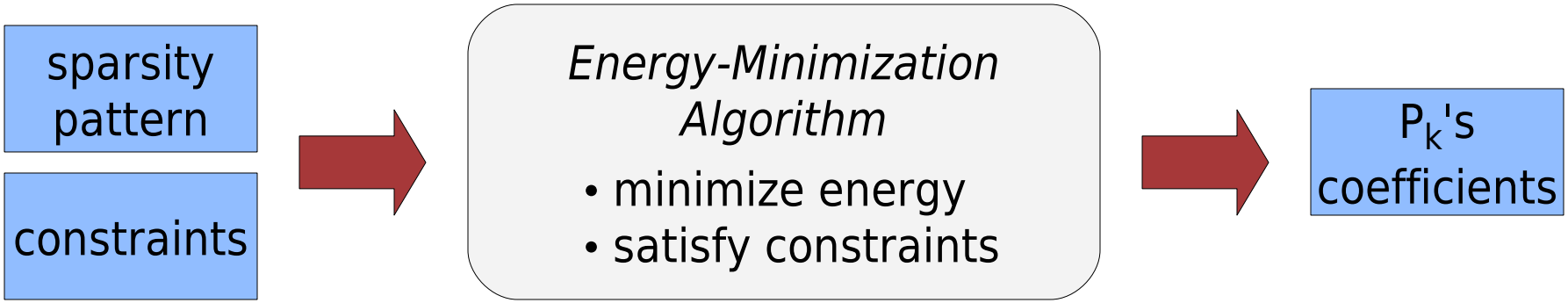
Algorithmic flexibility

Energy-minimization: Algorithm

$min \sum p_i^T A p_i$ where p_i gives the i^{th} column of a prolongator

Idea: construct the grid transfer operator P by minimizing the energy of each column P_k while enforcing constraints (sparsity pattern and specified modes).

Input / output of energy-minimization algorithm:



Advantages:

- Flexibility (input):
  - arbitrary coarsening
  - accept any sparsity pattern (arbitrary basis function support)
  - enforce constraints: important modes requiring accurate interpolation
  - choice of norm for minimization and search space
- Robustness

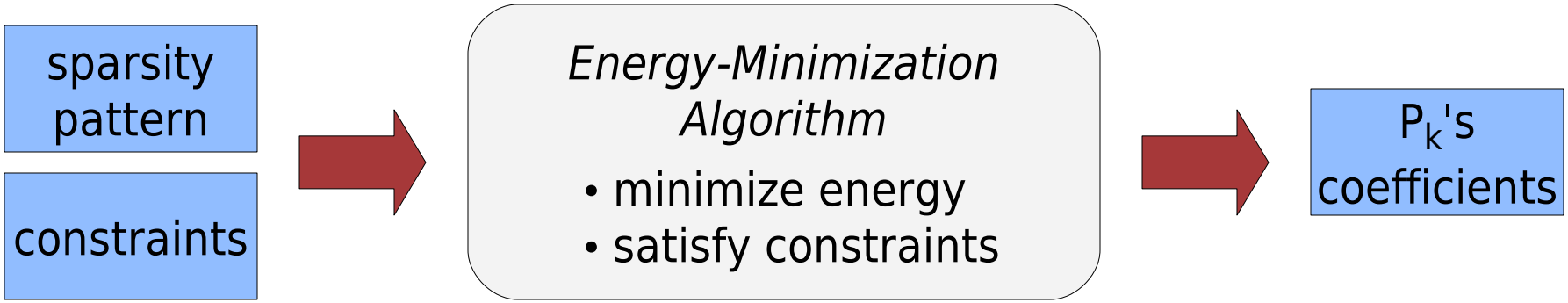
Sandia National Laboratories

Energy-minimization: Algorithm

$min \sum p_i^T A p_i$ where p_i gives the i^{th} column of a prolongator

Idea: construct the grid transfer operator P by minimizing the energy of each column P_k while enforcing constraints (sparsity pattern and specified modes).

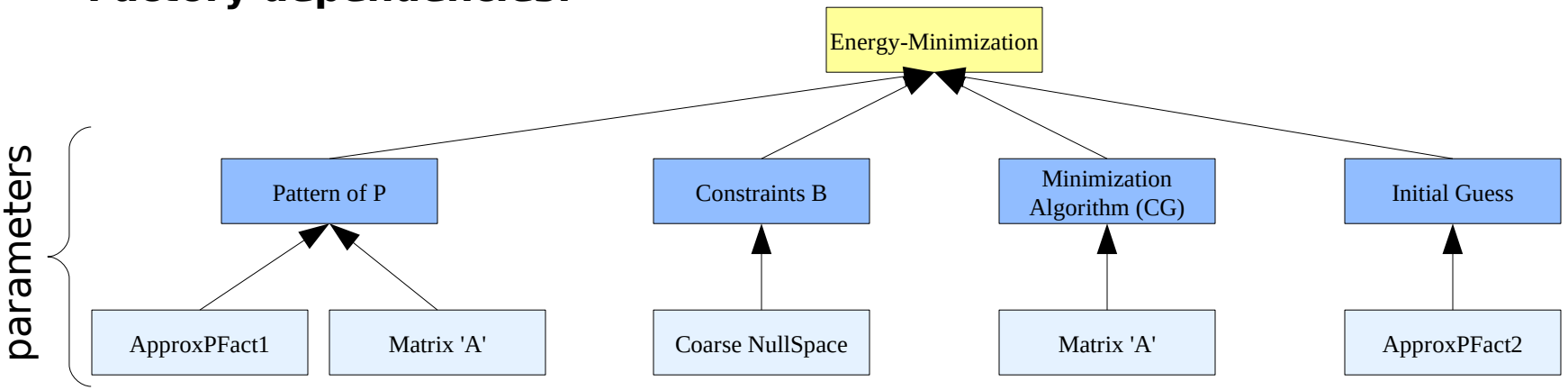
Input / output of energy-minimization algorithm:



Brannick, Brezina, Chan, Kolev, Mandel, Olson, Schroder, Smith, Vanek, Vassilevski, Wagner, Wan, Xu, Zikatanovc

Energy-Minimization

Factory dependencies:



Example:

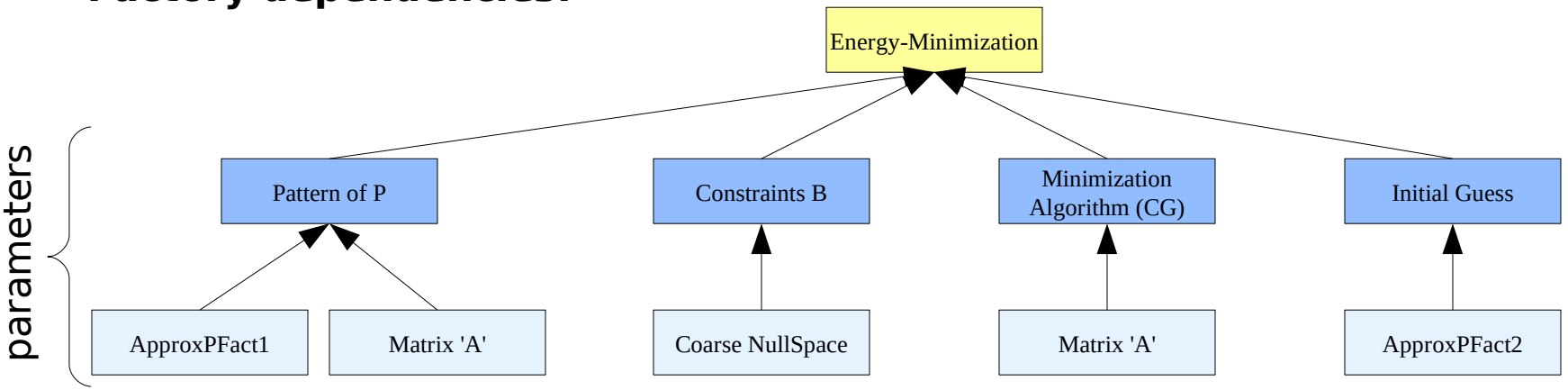
```
ApproxPFact = TentativePFactory();
PatternFact = PatternFactory(ApproxPFact);
ConstraintFact = ConstraintFactory();
EminSolver = CGEminSolver(10);

Pfact = EminPFactory(PatternFact,ConstraintFact,
                     EminSolver, ApproxPFact);
```

Sandia National Laboratories

Energy-Minimization

Factory dependencies:



Remarks:

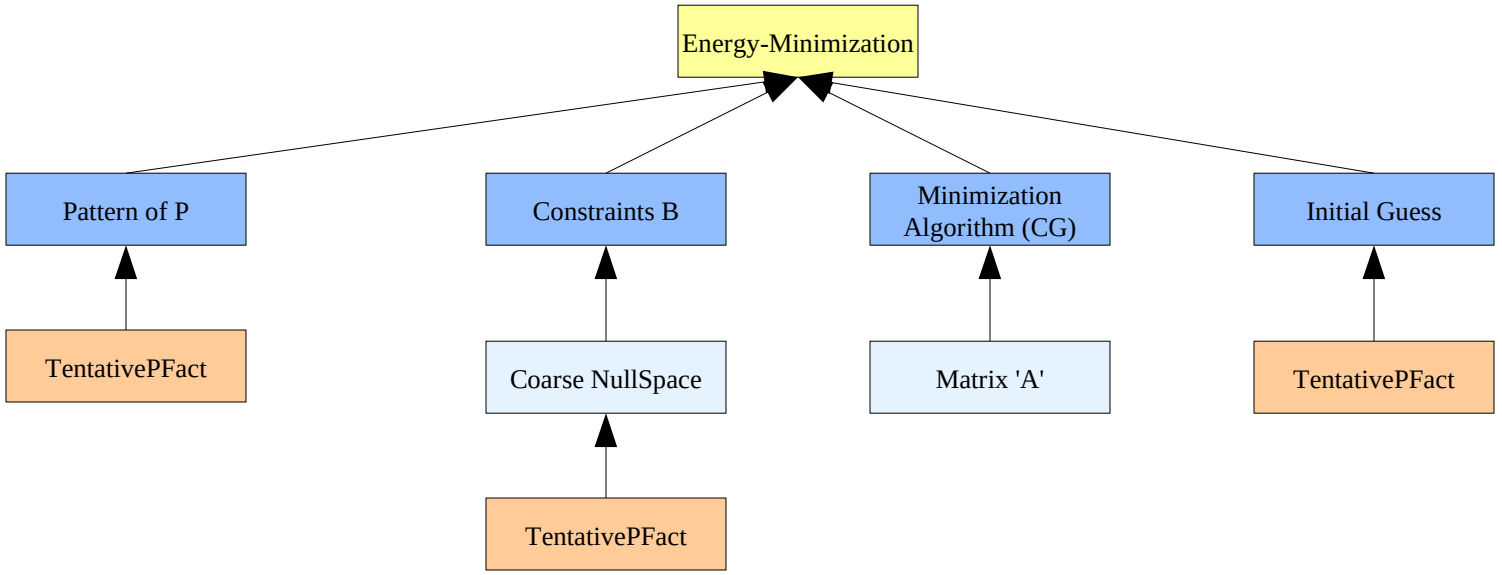
- ApproxPFact is used to build the initial guess of the Krylov minimization. The same factory can also be used to build the pattern of P
  Example of ApproxPFact: TentativePFactory
- Building the constraints matrix require the coarse nullspace. It can be built by different methods, like injection, but also by ApproxPFact.

Problems:

- Not intuitively obvious where different factories get plugged in.
- How to avoid recomputation when output of factories are reused?

Energy-Minimization

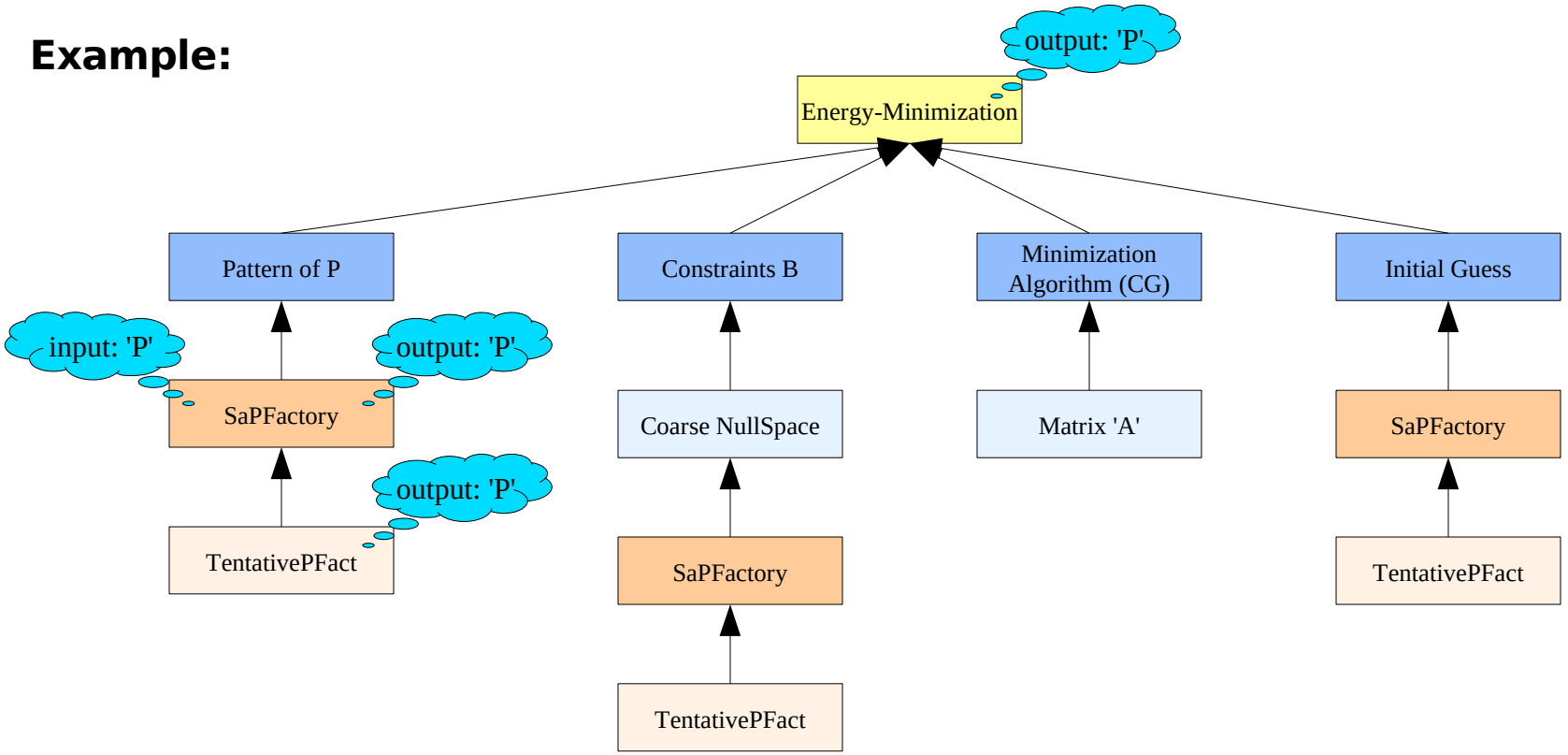
Example:



- What to do if a factory is used several time?
  - Idea 1: check if data already present on the hierarchy to avoid recomputation

Energy-Minimization

Example:



- What to do if a factory is used several time?
  - Idea 1: check if data already present on the hierarchy to avoid recomputation
  - Idea 2: on `Level`, keep track of which factory generates which data:

```
SaPFactory.Build(FineLevel, CoarseLevel) {
    Ptent = FineLevel('P', this.tentativePFact_);
}
```

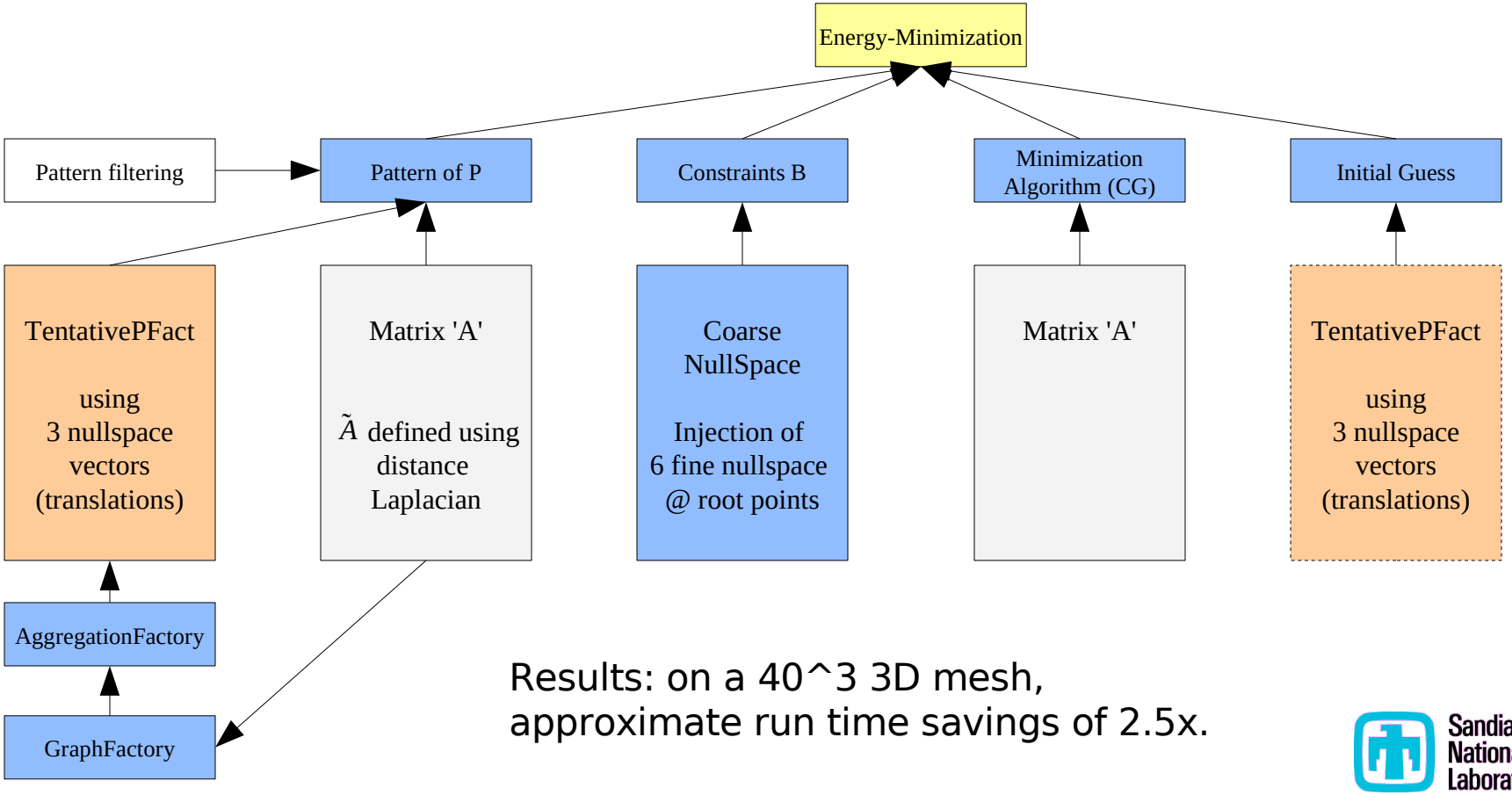
More advanced usage

Sandia National Laboratories

Exploiting the flexibility of the framework

Talk by Jonathan Hu - Friday morning: *Coarse Grid Representations of the Near Nullspace in an Energy Minimizing Multigrid*

Reduce # DOFs on the coarse grid for Elasticity 3D problems but enforce a good coarse grid representation of the 6 nullspace vectors:



Results: on a 40^3 3D mesh, approximate run time savings of 2.5x.

Sandia National Laboratories

Exploiting the flexibility of the framework

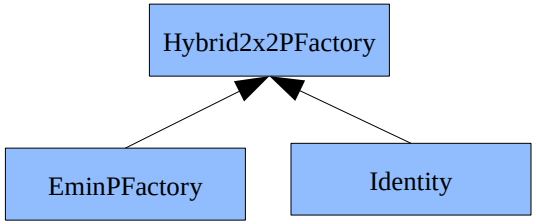
- Talk by Badri Hiriyur – was Tuesday morning: *A quasi-algebraic multigrid approach based on Schur complements for linear systems associated with XFEM*

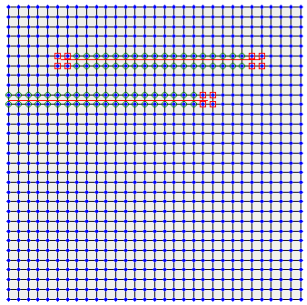
XFEM Linear System:

$$\begin{bmatrix} A_{rr} & A_{rx} \\ A_{xr} & A_{xx} \end{bmatrix} \begin{bmatrix} u_r \\ u_x \end{bmatrix} = \begin{bmatrix} \tilde{f}_r \\ \tilde{f}_x \end{bmatrix}$$

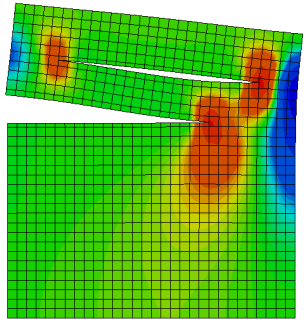
  - AMG applied on the Schur complement without explicitly forming it (specific prolongation and smoother factories are needed but reused existing capabilities)

$$\bar{P} = \begin{bmatrix} P & 0 \\ 0 & I \end{bmatrix}$$





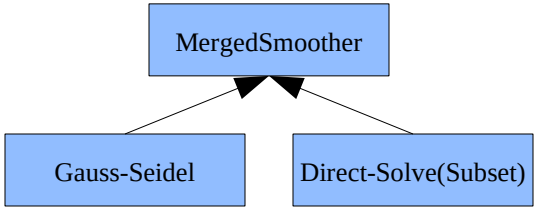
  - Aggregation & Sparsity pattern modified to maintain discontinuities on coarse levels
  - Crack geometry information projected on coarse levels
  - Only regular DOFs are currently coarsened



Sandia National Laboratories

Exploiting the flexibility of the framework

- Talk by Axel Gerstenberger – was Monday morning: *Algebraic Multi-Grid techniques for the eXtended Finite Element Method*
  - *Use a modified version of the nullspace for fracture problem*
  - *On the fine level, the smoother is a combination of classic smoothers on the whole system and a direct solve on enriched nodes*



- Talk by Tobias A. Wiesner – was Tuesday morning: *An improved AMG transfer operator for nonsymmetric positive-definite systems*
  - Reusing capabilities developed for symmetric MG to build appropriate nonsymmetric multigrid transfer operators
  - Adding prolongator filtering capabilities to the framework

Summary

- Flexible AMG framework helps address variety of scenarios.

- Energy minimization AMG is one path to flexibility:

  Basic ingredients:

  - Sparsity pattern
  - Interpolation constraints
  - Initial guess
  - Matrix to define energy (e.g. A)
  - Minimization algorithm with a preconditioning strategy

- Current status of software packages:
  - MueMat: will be released soon
  - MueLu: in development