An Architecture for Composing Solvers
Case study of PETSc

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
Outline

- Composing solvers
- Easing the burden on users in composing solvers
PETSc Fundamentals

- Options database - handles runtime options
- Vec - for storing field variables
- Mat - for storing (or applying) linear operators
- PC - for applying preconditioners
- KSP - for applying Krylov methods
- SNES - for solving nonlinear systems.
Vec Interpretation

- Abstract - element of a vector space
  - addition $z = x + y$
  - scalar multiplication $z = a^*x$
  - application of linear operator $z = Lx$
- As one dimensional array
  - VecGetArray(Vec,double*)
- In user code
  - DAVecGetArray(Vec,double**)
  - etc.
PC One-step Solvers

\[ x^{\text{new}} \leftarrow x + B(b - Ax) = x + BA(x^* - x) = x + Be \]

Gauss-Seidel (smoothing)

\[ = x + (L + D)^{-1}(b - (L + D + U)x) = (L + D)^{-1}(b - Ux) \]

Incomplete LU factorization

\[ = x + \hat{U}^{-1}\hat{L}^{-1}(b - Ax) \]

Full LU factorization
KSP Multi-step Solvers

Given

\[
x^{n-1}, x^n, \hat{x}^{n+1} = x^n + B(b - Ax^n)
\]

compute

\[
x^{n+1} = \alpha x^{n-1} + \beta x^n + \gamma \hat{x}^{n+1}
\]

inexpensively that is better in some measure.

- Conjugate gradient method
- Generalized minimum residual
- Quasi-minimal residual
- .......

Black-box on A, B and x
Composition of Preconditioners

A linear operator that improves an approximate solution to a linear system.

\[ x \leftarrow x + B(b - Ax) = x + BA(x^* - x) = x + Be \]

Constructing a preconditioner from two preconditioners.

\[ y \leftarrow x + B_1(b - Ax) \]
\[ x \leftarrow y + B_2(b - Ay) \]

Multiplicative version

\[ x \leftarrow x + (B_1 + B_2 - B_2AB_1)(b - Ax) \]

Additive version

\[ x \leftarrow x + (B_1 + B_2)(b - Ax) \]
Composition of PCs

KSPGetPC(ksp,&pc)
PSetType(pc,PCCOMPOSITE)
PCompositeSetType(pc,additive or multiplicative)
PCompositeAddPC(pc,PCJACOBI)
PCompositeAddPC(pc,PCSOR)
...

PCompositeAddPC(pc,PCKSP)
PCompositeGetPC(pc,2,&inksp)
KSPGetPC(inksp,&inpc)
PSetType(inpc,PCICC)
...

- pc_type composite
- pc_composite_type additive, multiplicative
- pc_composite_pcs jacobi, sor, ...
Preconditioners Defined by (near) Galerkin Process

Define restriction and interpolation operators

\[ R_i \] maps from a right hand side to a smaller, weighted right hand side.

\[ R_i^T \] interpolates from a subspace of the solution space to the entire solution space.

Define preconditioners by

\[ B_i = R_i^T (R_i A R_i^T)^{-1} R_i \quad B_i = R_i^T S_i R_i \]

Special cases - \( R_i \) has a single 1 per row, \( R_i A R_i^T \) is a submatrix of \( A \)

- **overlapping Schwarz methods** - selects all unknowns in a local domains
- **field split methods** - selects the \( i \)th component at each grid point
- **multigrid** - maps to a coarse grid
Composition of Galerkin Preconditioners

\[ y \leftarrow R_1^T S_1 R_1 (b - Ax) \]

\[ x \leftarrow y + R_2^T S_2 R_2 (b - Ay) \]

### Multiplicative

\[ x \leftarrow x + (R_1^T S_1 R_1 + R_2^T S_2 R_2) (b - Ax) \]

### Additive
Galerkin-like PCs

PCSetType(pc,PCGALERKIN)
PCGalerkinSetRestriction(pc,Mat R)
PCGalerkinGetKSP(pc,&gksp)
KSPSetOperators(gksp,Mat A,...)
Jacobi and Gauss-Seidel as Galerkin Composites

\[ B_i = R_i^T (R_iAR_i^T)^{-1} R_i \]

\[ R_i = [0, 0, ..., 1, 0, ...0] \quad R_iAR_i^T = A_{ii} \]

\[ B_i = [0, 0, ..., 1, 0, ...0]^T A_{ii}^{-1} [0, 0, ..., 1, 0, ...0] \]
FieldSplit Preconditioners

- Handle different components differently

- Example:

  -pc_type composite
  -pc_composite_pcs fieldsplit,sor
  -sub_0_pc_fieldsplit_0_fields 0
  -sub_0_fieldsplit_0_pc_type cholesky
Another FieldSplit Example

-pc_type fieldsplit
-sub_0_pc_fieldsplit_0_fields 0
  -sub_0_fieldsplit_0_pc_type hypre
  -sub_0_fieldsplit_0_pc_hypre_type boomeramg
-sub_1_pc_fieldsplit_1_fields 1
  -sub_1_fieldsplit_1_pc_type sor
Yet Another Example

- pc_type fieldsplit

- sub_0_pc_fieldsplit_0_fields 0,1
  - sub_0_fieldsplit_0_pc_type ksp
    - ksp_sub_0_fieldsplit_0_pc_type fieldsplit

  .......

- sub_1_pc_fieldsplit_1_fields 2
  - sub_1_fieldsplit_1_pc_type sor
Multigrid Fundamentals

- "Smoother" removes "high-frequency" error
- Stagnates on low-frequency error

\[ x^{\text{new}} \leftarrow x + B(b - Ax) = x + Be \]

\[ R_c^T \quad A_c^{-1} \quad R_c e \]

\[ R_c = R_{c+1} \rightarrow_c R_{c+1} \]
Multigrid

A specific set of techniques for composing Galerkin-like preconditioners (simple solvers).

Uses recursion, as an optimization, for efficient implementations.
Additive Multigrid

\[ \sum_{c} R_c^T S_c^{-1} R_c \]

- \texttt{pc\_mg\_type} additive

\[ \sum_{c} R_c^T \sum_{i} R_{c_i}^T A_{c_{ii}}^{-1} R_{c_i} R_c \]

- \texttt{mg\_levels\_pc\_type} jacobi
- \texttt{mg\_levels\_ksp\_type} preonly
- \texttt{mg\_levels\_pc\_type} sor
Additive

Multiplicative
Multigrid
- pc_mg_type multiplicative
- pc_mg_smoothdown 0, 1, 2, ...
- pc_mg_smoothup 0, 1, 2, ...
- pc_mg_cycles 1 or 2
Controlling the Smoothers

-mg_levels_ksp_type preonly, cg, gmres, ...
-mg_levels_1_pc_type icc, ...

-mg_coarse_pc_type redundant
-mg_coarse_redundant_type lu, ...

MGGetSmooother[Up, Down](pc, l, &lksp)
KSPSet....(lksp, .....)
Setting the Operators

```c
for (l=0; l<nlevels-1; l++) {
    MGGetSmoother(pc,l,&lksp)
    KSPSetOperators(lksp,A[l],A[l],SAME_N...
    ....
    MGSetInterpolation(pc,l,P[l])
    MGSetRestriction(pc,l,R[l])
```

\[-pc\_mg\_galerkin\ or\]

\[-\text{PCMGSetGalerkin}(pc)\]

\[
A_c = R A_{c+1} R^T
\]

```c
for (l=0; l<nlevels-1; l++) {
    MGSetInterpolation(pc,l,P[l])
    MGSetRestriction(pc,l,R[l])
```
Seems Pretty Easy?

- Must provide: interpolation/restriction
- Must provide: fine grid operator
- Can/should provide: level operators
- Can provide alternative operators on split fields
DMMG

Software to fill in the PCMG slots and manage the family of Mats, interpolations, field operators etc.

• Your code does not provide the Jacobians, partial Jacobians etc to the library.

• The library calls your code to compute these items as needed.

This allows all decisions about solver composition to be made at runtime!
DMMG Example

extern FormJacobian(..., Vec, Mat A, Mat B, ctx)

SNESSetJacobian(snes, A, B, FormJacobian, ctx);

extern FormJacobians(..., Grid, Vec, Mat A, Mat B, ctx)

DMMGSetSNES(snes, ..., FormJacobians, ctx);

FormLocalJacobians(..., LocalGrid, double**, Mat A, ...)

DMMGSetSNESLocal(snes, ..., FormLocalJacobians, ctx);
DA - Managing Structured Grids

DACreate2d(MPI_Comm, DA_NONPERIODIC, DA_STAR_STENCIL, Mp, Np, M, N, dof, stencil_width)

DACreateLocalVector(da, &vec)
DACreateGlobalVector(da, &vec)
DAGetMatrix(da, &mat)
DAGetInterpolation(dac, daf, &R)
DARefine(da, &daf)

DAGlobalToLocalBegin/End(da, gvec, lvec)
VecPack - Managing Collections of Grids and Free variables

VecPackCreate(MPI_Comm,&pack)
VecPackAddDA(pack,da1)
VecPackAddDA(pack,da2)
VecPackAddArray(pack,n)

.....

VecPackGather(pack,gvec,lvec1,....)
VecPackScatter(pack,gvec,lvec1,....)
Composition of solvers is becoming more important as more multi-physics simulations are being done.

Solvers must be composable.

Solvers should compose themselves.