Abstract Parallel Multigrid Software in PETSc 2.0

Barry Smith

http://www.mcs.anl.gov/petsc/petsc.html

http://www.mcs.anl.gov/home/bsmith

PETSc 2.0 is developed by
Satish Balay, Lois Curfman Mcinnes, Bill Gropp,and Barry Smith.

ARGONNE NATIONAL LABORATORY • UNIVERSITY OF CHICAGO •

PETSc: Portable, Extensible Toolkit for Scientific computation

- Ease the development of large-scale PDE application codes
- Provide infrastructure for rapid algorithm prototyping
- Create a programming environment that serves
  - Both black-box and high-performance users
  - Software developers
  - Algorithm researchers
- Support users in Fortran, C and C++

PETSc 2.0: The End Result

- Coded in ANSI C (also compiles with C++ compilers).
- All parallel communication is done using MPI.
- Uses other libraries (LAPACK, SPARSPAK, BlockSolve95, ESSL) where possible.
- Uses object oriented techniques
  - data encapsulation
  - polymorphism
  - inheritance from base abstract classes.
  - dynamic (run-time binding) of all "large" functions
- Uses a limited number of abstract concepts (base classes).

Levels of Abstraction in Numerical Software

- **Application-specific interface**: Programmer manipulates objects associated with the application.
- **High-level mathematics interface**: Programmer manipulates mathematical objects, such as PDEs, boundary conditions, domains.
- **Algorithmic and discrete mathematics**: Programmer manipulates mathematical objects like sparse matrices, nonlinear equations, as well as algorithmic objects such as solvers.
- **Low-level computational kernels**: for example, BLAS-type operations.

Philosophy

Multigrid - large family of algorithms. Rich set of different relationships between the algorithms.

Distributed memory parallel computers - complicated to program. Difficult to get good performance on.

Conclusion

There is a need to determine commonality through abstraction among multigrid algorithms and use this abstraction in their implementation.

"Abstract Multigrid", *Craig Douglas*

Abstraction already very successfully used in multigrid analysis.

Abstract Multigrid

- Useful approach in writing general purpose sequential multigrid codes.
- Almost required in handling the much more complicated case of distributed memory parallel computers.

Components in a Multigrid Code

Consider only **linear** problems with **no mesh refinement**.

- “Smoother”
- Coarse grid “solver”
- Compute residual
- Interpolate (and add)
- Restrict
- Compute residual

Standard Combinations: V-cycle

```
If (level is coarsest) then solve coarse grid problem
else {
  Smooth
  Compute residual
  Restrict the residual to the coarser grid
  Call V-cycle on coarser grid
  Interpolate (and add)
  Smooth
}
```

Standard Combination: Additive

Restrict fine grid residual to each grid

On each grid smooth (or solve)

Interpolate and add the results to the finest grid

Standard Combination: Full (a variant)

```
Restrict fine grid residual to each grid

From coarsest to finest grid {
   Call V-cycle on grid
   Interpolate solution to next grid
}
Call V-cycle on finest grid
```

Standard Combination: Kaskade (a simple variant)

```
Restrict fine grid residual to each grid

From coarsest to finest grid {
   Smooth (or solve) on grid
   Interpolate solution to next (finer) grid
}
Smooth on finest grid
```

Organization of the Software

Mathematical objects (i.e. linear operator)	Software variable (object) (Mat)
Simple algorithm (basic kernel)	Subroutine
Complete algorithm (multigrid)	Subroutine + context variable (Explained below)

Variables as Objects

Collection of the **data** and **functions** that act upon that data.

Uses **data encapsulation** and **polymorphism** so that, for example, MatMult(A,x,y) calls the correct matrix vector product routine independent of the format of A.

PETSc Objects Required for Multigrid

- `Vec` - vectors
- `Mat` - abstract matrices (includes support for matrix free methods)
- `SLES` - linear solvers (including smoothers)
  - `PC` - preconditioners (stationary linear solvers)
  - `KSP` - Krylov subspace methods (including Richardson)
- `MG` - multigrid context

Mat Object

Abstract linear operator. Useful for interpolation (and add), restriction and residual computation.

- compressed row storage
- block compressed row storage
- block diagonals storage
- user provided function

PC Object (stationary linear solvers)

- Jacobi
- SOR (per processor, and global)
- ILU, ICC
- block Jacobi
- ASM (overlapping additive Schwarz)

Block solvers can use any SLES solver on each block.

MG (Multigrid) Object

For each level contains

- `SLES` - smoothers and coarse grid solvers
- `Mat A` - linear operator at that level
- `Mat I` - interpolation operator
- `Mat R` - restriction operator
- `Vec x,b,r` - work vectors

Context Object

Variable that contains all the information required for the execution of an algorithm,

- required data
- code (sometimes)

Actual C Structure for MG

```
struct _MG
{
    MGType    am;                     /* Multiplicative, additive, ... */
    int       cycles;                 /* Number cycles to run */
    int       level;                  /* level = 0 coarsest level */
    Vec       b;                      /* Right hand side */
    Vec       x;                      /* Solution */
    Vec       r;                      /* Residual */
    int       (*residual)(Mat,Vec,Vec,Vec);
    Mat       A;                      /* matrix used in forming residual*/
    SLES      smoothd;                /* pre smoother */
    SLES      smoothu;                /* post smoother */
    Mat       interpolate;
    Mat       restrct;
    SLES      csles;                  /* coarse grid solve */
};
```

Putting it All Together: V or W-cycle

```
int MGMCycle_Private(MG *mglevels) {
  MG     mg = *mglevels, mgc = *(mglevels + 1);
  int    cycles = mg->cycles, ierr,its;

  if (mg->level == 0) {
    SLESSolve(mg->csles,mg->b,mg->x,&its);
  }
  else {
    while (cycles--) {
      SLESSolve(mg->smoothd,mg->b,mg->x,&its);
      (*mg->residual)(mg->A, mg->b, mg->x, mg->r );
      MatMult(mg->restrct,  mg->r, mgc->b );
      VecSet(&zero,mgc->x);
      MGMCycle_Private(mglevels + 1);
      MatMultTransAdd(mg->interpolate,mgc->x,mg->x,mg->x);
      SLESSolve(mg->smoothu,mg->b,mg->x,&its);
    }}
  return 0;}
```

Putting it All Together: Additive

```
int MGACycle_Private(MG *mg)
{
  int    i, l = mg[0]->level,its,ierr;

  for ( i=0; i<l; i++ ) {
    MatMult(mg[i]->restrct,  mg[i]->b, mg[i+1]->b);
  }
  for ( i=0; i<l; i++ ) {
    VecSet(&zero,mg[i]->x);
    SLESSolve(mg[i]->smoothd, mg[i]->b, mg[i]->x,&its);
  }
  VecSet(&zero,mg[l]->x);
  SLESSolve(mg[l]->csles, mg[l]->b, mg[l]->x,&its);
  for ( i=l-1; i>-1; i-- ) {
    MatMultTransAdd(mg[i]->interpolate,mg[i+1]->x,mg[i]->x,mg[i]->x);
  }
  return 0;}
```

What Does This Have to Do with Parallelism?

It decouples the multigrid algorithm from the particular type machine it is running on.

Provides the easy ability to use any combination of smoothers and multigrid varients.

Allows individual kernels (smoothers, etc.) to be optimized independently of the underlying multigrid algorithm.

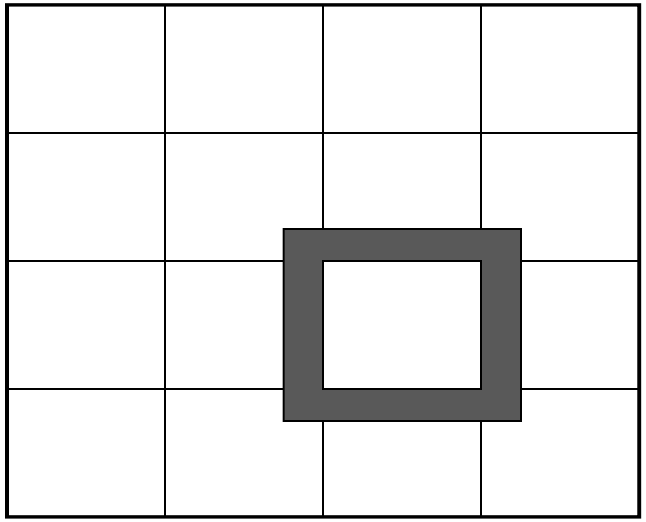
Allows high performance parallel multigrid codes to be (relatively) easily written.

Example: Two Level Method Inside Newton Solver

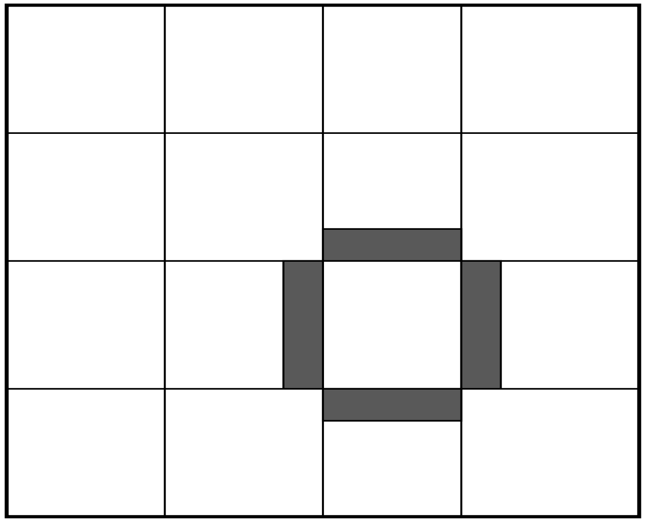
$$
\begin{aligned}
-u_{xx} - u_{yy} - \lambda e^{u} &= 0 && \text{on the unit square} \\
u(x,y) &= 0 && \text{on the boundary}
\end{aligned}
$$

$$u_{xx} \approx (u_{i+1j} - 2u_{ij} + u_{i-1j})/h.$$

Distributed Array To Handle Function Evaluations



Box-type stencil



Star-type stencil

```
DACreate2d(MPI_Comm comm,DA_STENCIL_BOX,DA_NONPERIODIC,
           int M, int N, int m, int n,int w, int s, DA *da);
DACreate2d(MPI_Comm comm,DA_STENCIL_STAR,DA_XYPERIODIC,
           int M, int N, int m, int n, int w, int s, DA *da);
```

Flow of the Code

User provides all of the parameters for the nonlinear solve.

- → SNESSolve(snes,x)
  - → ComputeFunction() - Generates a new b.
  - → ComputeJacobian() - Generates new A_f and A_c.
  - → SLESSolve(sles,...)
    - → MG-Cycle(...)
      - → SLESSolve(sles_fine,...)
        - → Jacobi(A_f,..)
      - → MatMult(Restriction,...)
      - → MG-Cycle(...)
        - → SLESSolve(sles_coarse,...)
          - → LUSolve(A_c,...)
      - → MatMultTransAdd(Restriction,...)

Application Data Structure

```
#include "snes.h"
#include "da.h"
#include "mg.h"
#include <math.h>
#include <stdio.h>

typedef struct {
   double      param;           /* test problem parameter */
   int         mx,my;           /* fine mesh in x, y directions */
   int         Mx,My;           /* coarse mesh in x, y directions */
   Vec         localX,localF;   /* ghosted local vector */
   DA          da;              /* distributed array data structure */
   Mat         J_coarse,J_fine; /* Jacobians on the two grids */
   SLES        sles_coarse;
   int         ratio;
} AppCtx;
```

Application Setup Code

```
/* Create nonlinear solver */
SNESCreate(MPI_COMM_WORLD,SNES_NONLINEAR_EQUATIONS,&snes);

/* provide user function and Jacobian */
SNESSetFunction(snes,r,FormFunction,(void *)&user);
MatCreate(MPI_COMM_WORLD,n,n,&user.J_fine);
SNESSetJacobian(snes,user.J_fine,user.J_fine,FormJacobian,&user);

/* set two level additive preconditioner */
SNESGetSLES(snes,&sles);
SLESGetPC(sles,&pc);
PCSetType(pc,PCMG);
MGSetLevels(pc,2);
MGSetType(pc,MGMULTIPLITIVE);
```

Application Setup Code (continued)

```
/* Create coarse level */
MGGetCoarseSolve(pc,&user.sles_coarse);
MatCreate(MPI_COMM_SELF,N,N,&user.J_coarse);
SLESSetOperators(user.sles_coarse,user.J_coarse,user.J_coarse,...);

MGSetX(pc,COARSE_LEVEL,x_coarse);
MGSetRhs(pc,COARSE_LEVEL,r_coarse);

/* Create fine level */
MGGetSmoother(pc,FINE_LEVEL,&sles_fine);
SLESSetOperators(sles_fine,user.J_fine,user.J_fine,...);
```

Function Evaluation Code

```
int FormFunction(SNES snes,Vec X,Vec F,void *ptr)
{
  ....

  /* Get ghost points */
  DAGlobalToLocalBegin(user->da,X,INSERT_VALUES,localX);
  DAGlobalToLocalEnd(user->da,X,INSERT_VALUES,localX);
  VecGetArray(localX,&x);
  VecGetArray(localF,&f);
  DAGetCorners(user->da,&xs,&ys,0,&xm,&ym,0);
  DAGetGhostCorners(user->da,&Xs,&Ys,0,&Xm,&Ym,0);
```

Function Evaluation Code (continued)

```
for (j=ys; j<ys+ym; j++) {
  row = (j - Ys)*Xm + xs - Xs - 1;
  for (i=xs; i<xs+xm; i++) {
    row++;
    if (i == 0 || j == 0 || i == mx-1 || j == my-1 ) {
      f[row] = x[row];
      continue;
    }
    u = x[row];
    uxx = (two*u - x[row-1] - x[row+1])*hydhx;
    uyy = (two*u - x[row-Xm] - x[row+Xm])*hxdhy;
    f[row] = uxx + uyy - sc*exp(u);
  }
}
/* Insert values into global vector */
DALocalToGlobal(user->da,localF,INSERT_VALUES,F);
```

Jacobian Evaluation Code: Fine Grid

```
int FormJacobian_fine(AppCtx *user,Vec X,Mat *J,Mat *B,MatStructure *f)
{
  ....
  for (j=ys; j<ys+ym; j++) {
    row = (j - Ys)*Xm + xs - Xs - 1;
    for (i=xs; i<xs+xm; i++) {
      row++; grow = ltog[row];
      v[0] = -hxdhy; col[0] = ltog[row - Xm];
      v[1] = -hydhx; col[1] = ltog[row - 1];
      v[2] = two*(hydhx + hxdhy) - sc*lambda*exp(x[row]); col[2] = grow;
      v[3] = -hydhx; col[3] = ltog[row + 1];
      v[4] = -hxdhy; col[4] = ltog[row + Xm];
      MatSetValues(jac,1,&grow,5,col,v,INSERT_VALUES);
    }
  }
  *f = SAME_NONZERO_PATTERN;
```

Conclusion

Having an abstract multigrid infrastructure allows one to quickly write complete parallel multigrid algorithms by providing application specific routines.

The initial learning curve for this approach is high, but once the threshold is reached, new application codes are easily written.

PETSc 2.0 is freely available

http://www.mcs.anl.gov/petsc/petsc.html

ftp://info.mcs.anl.gov/pub/petsc/

PETSc is funded by the United States Department of Energy and the National Science Foundation.